University of Alberta

LEARNING TO IMPROVE QUALITY OF THE PLANS PRODUCED BY PARTIAL ORDER PLANNERS

by

Muhammad Afzal Upal ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 2000

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-59687-7

Canada

University of Alberta

Library Release Form

Name of Author: Muhammad Afzal Upal

Title of Thesis: Learning to Improve Quality of the Plans Produced by Partial Order Planners

Degree: Doctor of Philosophy

Year this Degree Granted: 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Afzal Upal

Muhammad Afzal Upal
87 Grosvenor Cres
Saskatoon, SK
Canada, S7J 2S1

Date: June 1, 2000.

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Learning to Improve Quality of the Plans Produced by Partial Order Planners** submitted by Muhammad Afzal Upal in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in *Computing Science.*

Renée E Elio

Renee Elio (Supervisor)

R. Goebel

Randy Goebel

Robert Mercer

Eleni Stroulia

Roger W Toogood

Roger Toogood

Date: June 14, 2000

Abstract

Generally speaking, AI planning research can be divided along two lines: domain independent planning and practical planning. While domain independent planning work has mostly focussed on building general, systematic and complete planners, practical planning work has been driven by concerns such as planning efficiency and plan quality. The result is that while domain independent planners have many desirable theoretical properties (such as *completeness*), they are too inefficient to use in any real world situation. Practical planners on the other hand can produce high quality solutions for the real world problems but are not general enough to be applied to any domain other than the one they were built for. Machine learning for planning aims to bridge this gap by building planning and learning systems that can learn domain specific knowledge that can help them efficiently produce high quality solutions. While considerable planning and learning research has been done to learn to improve planning efficiency, little work has been done to learn to improve plan quality, especially in the context of the newer partial-order planners. But AI planners must learn to produce high quality solutions if they are to be deployed in the real world situations. This work addresses the problem of learning to improve plan quality for partial-order planners. It presents a planning and learning framework called PIP (Performance Improving Planner) that learns domain specific quality improving heuristics by comparing two planning episodes of different quality to identify the planning decisions that result in the higher quality plan. Empirical results on a number of benchmark as well as artificial planning domains show that the PIP approach leads to efficient production of high quality plans. PIP's learning algorithm is also analyzed as a supervised concept learner that learns to discriminate between the partial plans it

P

encounters during the search to learn to apply the appropriate planning decisions (i.e., the planning decisions that will lead towards the generation of higher quality plans). The ideas and results of this work contribute to the development of AI planning systems for problems where plan quality is an important concern.

To my mother who taught me that I could achieve anything I set my mind to.

Acknowledgements

Thanks to my supervisor Reneé Elio for all her help, support and guidance. Thanks to my wife for putting up with me when I was depressed and wanted to get away from it all. I am also grateful to my external examiner for his extensive comments and helpful suggestions.

I am also thankful to all the people who welcomed me to the Department of Computer Science, to the University of Alberta, and to the city of Edmonton and made each place feel like home. To Srinivas Padmanabhuni for his guidance and help. To Zameer Chaudhary and Naseer Chaudhary for welcoming me to their homes, for driving me around town and for the frequent dinner invitations. To Andreas Junghans and David McCoughan for being such good listeners. To Dmitri Gorodnichi for playing Soccer in GSB 705. All of you made me feel that I belonged.

But most of all I need to thank Julian Fogel for the intellectual conversations, for tutorials on logic, for the emotional support, and for always being there whenever I needed him. If there is any one person without whose help I cannot imagine having made through it all, it is you.

Contents

1 **Introduction** 1
- 1.1 Plan Quality 4
- 1.2 Problem Description 6
- 1.3 Contributions Of This Work 7
- 1.4 Organization Of This Dissertation 8

2 **Background** 10
- 2.1 The AI Planning Problem 10
  - 2.1.1 Knowledge Representation 10
  - 2.1.2 Search Techniques 13
  - 2.1.3 Decision Theoretic Planning 17
- 2.2 Learning to Improve Planning Efficiency 19
  - 2.2.1 Inductive Learning Techniques 20
  - 2.2.2 EBL 21
  - 2.2.3 Case-based Learning 24
  - 2.2.4 Hybrid Techniques 25
- 2.3 Learning to Improve Plan Quality 26
  - 2.3.1 Plan Quality Measurement Knowledge 26
  - 2.3.2 Analytic Techniques for Learning to Improve Plan Quality 29
  - 2.3.3 Non-analytic Techniques for Learning to Improve Plan Quality 30
  - 2.3.4 Planning by Rewriting 31
- 2.4 Summary 34

3 **The PIP Framework** 36
- 3.1 Knowledge Representation Scheme 36
  - 3.1.1 Value Functions for Quality 36
  - 3.1.2 Representing and Reasoning with Resources 37
- 3.2 Architecture and Algorithms 40
  - 3.2.1 Step 1: Generating the Default Planning Episode . . . 42
  - 3.2.2 Step 2: Generating the Model Planning Episode 50
  - 3.2.3 Step 3: Analytically Comparing the two Episodes . . . 54
  - 3.2.4 Step 4: Forming and Storing Domain Specific Rules . . 61
- 3.3 Summary 68

4 PIP-rewrite 69
4.1 PIP-rewrite's Architecture and Algorithm 71
4.1.1 The Planning Component 71
4.1.2 The Analytic Learning Component 72
4.1.3 The Rule library 75
4.2 Comparison of Rewrite and Search Control Rules 83
4.2.1 Methodology 83
4.2.2 Domain Descriptions 87
4.2.3 Experimental Set-up 89
4.2.4 Results 90
4.2.5 Discussion 92
4.3 Summary 97

5 Evaluating PIP 99
5.1 Empirical Comparison With SCOPE 99
5.1.1 Experimental Set-up 100
5.1.2 Results 100
5.2 Analysis of Factors That Affect PIP's Performance 103
5.2.1 PIP's Learning Component, ISL, As A Supervised Concept Learner 103
5.2.2 Factors For Evaluating Supervised Learning Algorithms 103
5.2.3 Empirical Experiments Using Artificial Domains 104
5.2.4 Varying Instance Similarity 107
5.2.5 Varying the Quality Branching Factor 122
5.2.6 Varying the Correlation Between the Planner Biases and the Quality Improving Biases 125
5.3 Summary 128

6 Conclusions and Future Work 130
6.1 Major Contributions of This Work 131
6.2 Future Research Directions 133
6.2.1 Better Rule Organization 134
6.2.2 Extending PIP To Deal With More Expressive Languages 135
6.2.3 Combining ISL With EBL From Failures 136
6.2.4 Extending PIP's Techniques For Non-classical AI Planners 136
6.2.5 Extending PIP-rewrite's Techniques For Non Classical AI Planners 138
6.3 Summary 138

Bibliography 139

A PR-STRIPS Encoding of Transportation Domain 146

B PR-STRIPS Encoding of Softbot Domain 148

C PR-STRIPS Encoding of Manufacturing Process-planning Domain 150

D An Abstract Domain 153

List of Figures

1.1 The Transportation World. 4
1.2 A Transportation problem. 5

2.1 Veloso's logistics domain: a resource-less version of Transportation domain. 12
2.2 Part of the state-space search tree for the Logistics problem of Figure 1.2. 14
2.3 A logistics problem. 15
2.4 A planning example from Veloso's logistics domain. 20
2.5 Search-tree for the problem shown in Figure 2.4. 22
2.6 Search-control rule learned by SCOPE 23
2.7 Search-control rule learned by SNLP+EBL 23
2.8 A search control and a rewrite rule learned from the same opportunity 32
2.9 A Softbot planning problem and two solutions for it. 32
2.10 A Transportation planning problem and two solutions for it. . 33
2.11 Part of the rewrite rule learned by PIP-rewrite from the training problem shown in Figure 2.10 34

3.1 PIP's architecture. 41
3.2 PIP's high level algorithm. 42
3.3 Problem 1: A Transportation planning domain. 43
3.4 The POP algorithm. 45
3.5 Continuation of the POP algorithm. 46
3.6 Continuation of the POP algorithm. 47
3.7 Default planning trace for the transportation example problem (Problem 1). 48
3.8 Continuation of Figure 3.7. 49
3.9 Ordered constraint-set corresponding to the planning trace shown in Figures 3.7. 51
3.10 PIP's Infer-constraints algorithm. Comments are enclosed in square brackets. 53
3.11 Model constraint-set for Problem 1. 54
3.12 The Intra-Solution Learning (ISL) Algorithm (Step 3 of Algorithm 1). 55
3.13 Continuation of the Intra-Solution Learning (ISL) Algorithm. . 56

3.14 Conflicting choice point. 58
3.15 Two planning decision sequences identified by ISL for the first conflicting choice point shown in Figure 3.14. 61
3.16 Generalized planning decision sequences. 62
3.17 Search Control Rule 1. 64
3.18 Search Control Rule 2. 64
3.19 Problem 2: A Transportation planning problem. 65
3.20 A conflicting choice point where application of a rule leads to a lower quality plan. 67

4.1 Search Control Rule 1 and Search Control Rule 2. 70
4.2 Rewrite Rule 1 71
4.3 DerPOP's planning algorithm. 71
4.4 A Transportation problem. 74
4.5 Uninstantiated planning trace for the default plan shown in Figure 4.4. 75
4.6 Instantiated planning trace for the default plan shown in Figure 4.4. 76
4.7 Learning opportunities identified by ISL using the uninstantiated default trace shown in Figure 4.5. 77
4.8 Learning opportunities identified by ISL using the instantiated planning trace shown in Figure 4.6. 78
4.9 Problem 2: A Transportation planning problem. 79
4.10 DerPOP's plan for the problem shown in Figure 3.19. 79
4.11 Rule retrieved by PIP-rewrite. 79
4.12 The initial plan after the application of Rewrite Rule 1. 80
4.13 The refine algorithm of PIP-rewrite 81
4.14 Problem 3: A training problem drawn from the softbot. 94
4.15 Problem 4: Another problem drawn from the softbot domain. 95
4.16 Modified form of Search Control Rule 3. 96

5.1 Graph showing how PIP and SCOPE improve planning efficiency. 101
5.2 Graph showing how PIP and SCOPE improve plan quality. . . 101

6.1 Move-briefcase action from Pednault's Briefcase Domain. . . . 135
6.2 A planning graph. 137

List of Tables

4.1 Performance data for the process planning domain. 90
4.2 Rule data for the process planning domain in the 20-problem case. 90
4.3 Performance data for the transportation domain. 91
4.4 Rule data for the transportation domain in the 20-problem case. 91
4.5 Performance data for the softbot domain. 91
4.6 Rule data for the softbot domain in the 20-problem case. . . . 92

5.1 Mean plan quality metric as a function of problem similarity and training set size. 110
5.2 Mean planning efficiency metric as a function of problem similarity and training set size. 111
5.3 Mean and standard deviation of the proportion of the useful rules in the 20-problem case as a function of problem similarity. 111
5.4 Mean and standard deviation of the proportion of the rules needing refinement in the 20-problem case as a function of problem similarity. 111
5.5 Mean and standard deviation of the plan quality metric in the 20-problem case as a function of problem similarity. 112
5.6 Mean plan quality metric as a function of domain similarity and the training set size. 117
5.7 Mean planning efficiency metric as function of domain similarity and the training set size. 117
5.8 Mean and standard deviation of the proportion of useful rules in the 20-problem case as a function of domain similarity. . . . 117
5.9 Mean and standard deviation of the proportion of rules needing refinement in the 20-problem case as a function of domain similarity. 118
5.10 Mean and standard deviation of the plan quality metric in the 20-problem case as a function of domain similarity. 118
5.11 Mean planning efficiency metric as a function of quality branching factor and training set size. 124
5.12 Mean plan quality metric as a function of quality branching factor and training set size. 125
5.13 Mean and standard deviation of the plan quality metric in the 20-problem case as a function of quality branching factor. . . . 125

5.14 Mean plan quality metric as a function of bias correlation and training set size. 126
5.15 Mean and standard deviation of the plan quality metric in the 20-problem case as a function of bias correlation. 126
5.16 Mean planning efficiency metric as a function of bias correlation and training set size. 127

Chapter 1

Introduction

Most human activities are goal directed. Whether playing a game, planning a vacation or creating a business plan, we are constantly engaged in generating strategies to achieve various objectives. Planning is the cognitive activity of devising strategies to achieve such goals. The aim of Artificial Intelligence (AI) planning research is to build *planners*, computer systems that can automatically generate plans. Such systems can be extremely useful in complex real-world situations such as military logistics planning, manufacturing process planning, and physical and urban planning. It is no surprise then that planning received the attention of AI researchers from the very beginning. Newell and Simon [NS72] proposed the first computer model of human problem solving with their influential work on GPS (General Problem Solver). A lot has changed in AI since then, but their formulation of the planning problem has endured. Generally speaking, this formulation of the planning problem is as follows:

Given

- a world description that describes the current world state,
- a world description that describes the desired (goal) state,
- and domain knowledge that describes how different actions affect the world.

Find

- a series of actions that can transform the current world state into one in which the goal is true.

The sequence of actions is the plan that, when executed, achieves the goals. Although this simple formulation ignores many issues, such as dynamic environments, it characterizes planning as a search process: the search for actions that can be executed and that are relevant to achieving the goal.

Planning algorithms can be organized along two broad dimensions. One dimension is *state-space total-order* planners versus *plan-space partial-order* planners and the other is *domain independent* versus *domain dependent* planners. State-space planners search in the space of world-states while partial-order planners search in the space of plans. Each node in the space of plans is a *partial plan* that contains the actions that so far have been determined to be needed in the plan and some constraints on those actions. It has been shown that partial order planners are more efficient than state-space planners on many interesting types of domains because they need to backtrack less [BW94]. The crucial features of partial-order plan-space planners are discussed in more detail in later chapters. At this point, it is important only to note that most previous work on improving plan quality has concerned state-space planners.

The initial focus of most work on planning was on building domain independent planning systems. Such systems do not use any extra domain knowledge during the search process which allows them to be deployed in a new domain with minimum effort. Many working planning systems were built and shown to solve problems in a number of toy planning domains such as Blocksworld [AHT90]. The hope was that this modular approach could be scaled up to more complex domains. However, a number of negative computational complexity results were quickly obtained showing that domain independent planning is a very hard combinatorial problem [ENS95]. While the research on improving domain independent planning algorithms continues [KS98], even the most advanced of the domain independent planning systems take exponential time to solve practical planning problems and cannot even solve large problems from the toy domains [GS96].

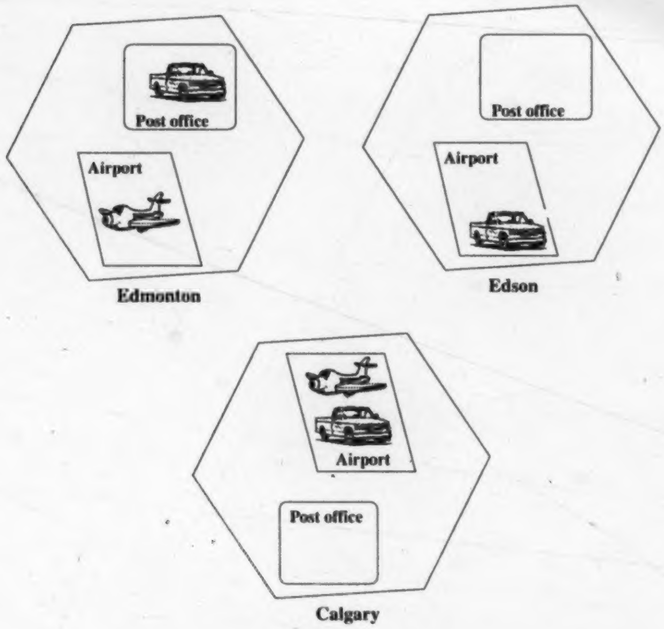
This has forced those interested in building practical planning systems to abandon this modular approach in favor of hard-coding domain knowledge into the search algorithm to solve real-world planning problems [Min89]. However, acquiring the domain knowledge and then encoding it into a form in which a planner can use it to limit search is a costly process in terms of the person hours required. Typically, it involves knowledge engineers interacting with domain experts for a considerable amount of time to elicit the domain knowledge and then encoding this knowledge into search heuristics. This manual process makes it very expensive to build efficient planning systems for real-world applications.

Machine learning for planning offers a possible solution by allowing a domain independent planning system to automatically (or semi-automatically) acquire search control knowledge to improve its planning performance. The basic idea is to add a machine learning module to a domain independent planning system so that the domain heuristics can be acquired automatically over some training period possibly eliminating the knowledge engineer (and sometimes the domain expert as well).

Various learning-to-plan systems have been designed over the last decade [Min89, Vel94]. The integrated planning and learning systems (sometimes called the *speed up learning systems*) learn domain-specific search control rules or remember past planning episodes to make the planning process more efficient. The learned knowledge helps greatly reduce backtracking by focusing the planner's attention on the choices that have, in the past, led to success in similar planning situations. However, even speed-up learning systems have had limited success in deployment to the real-world planning situations. One reason is that most planning and learning systems assume that any workable plan is good enough. This is often not the case in real-world planning situations where good quality plans are required.

1.1 Plan Quality

Classical planning and learning systems provide very limited representation of the plan quality knowledge. Essentially, most planning algorithms distinguish only between plans that fail to achieve the goal(s) and plans that succeed. This is very restrictive if plan quality is a real concern.



An object can be loaded/unloaded to/from a truck and it takes 5 minutes to do that and costs $5. An object can be loaded/unloaded to/from a plane and it takes 20 minutes and $15 to do that. It takes distance(A,B)/50 minutes to drive a truck from location A to location B and costs distance(A, B)/50 dollars. It takes distance(A, B)/1000 minutes to fly a plane from airport A to an airport B and costs distance(A, B)/5 dollars.

Figure 1.1: The Transportation World.

Consider the Transportation World shown in Figure 1.1. In this example the world consists of a number of cities shown by pentagons. Each city has two locations: an airport (shown as parallelograms) and a post office (shown as round-edged rectangles). Each city also has a truck that may be stationed at the post-office or the airport. An airport may also have a plane. Trucks can be used to travel between any two locations while a plane can only fly to an airport. In this world there are two ways of transporting objects from one location to another: using a plane or using a truck. A typical problem (such as the one shown in Figure 1.2) in the Transportation World is to find a plan to transport some objects from their current locations to some goal locations.

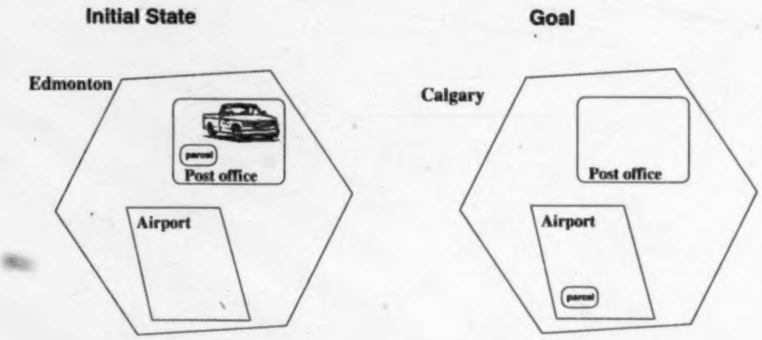


Figure 1.2: A Transportation problem. In the initial state an object *parcel* is at the Edmonton Post Office and the goal is to get it to Calgary Post Office

At first glance, this seems to be just the kind of problem addressed by the existing planning systems. And, indeed, given an encoding of simplified versions of this problem, existing planning systems are guaranteed to produce a viable plan if one exists. But the difficulty is that in this domain not all valid plans may be equally preferable. For instance, the plan to use a plane to transport the objects would take much less time but cost more in terms of money than the plan that uses a truck. Which plan is better depends on how much time and money it requires and and on how valuable time is with respect to money. For instance, if time is more important, then clearly the plan to fly the objects would be preferable over the plan to drive the objects.

The transportation example illustrates that the planning agent may also have preferences about other aspects of the world apart from the goals, such as consumption of resources. Similar situations arise in other domains as well. Imagine a softbot [Wil96] with the goal of finding a particular person's email address. A planning agent for this task may have three alternative courses of action to consider: search the net looking for some online staff directory, to send email to a mutual friend who may know the email address of the person, or to hire an online detective agency that charges $100 for the service of finding someone's email address, phone number and home address. Again which course of action is preferred depends upon the resource consumption by each plan and on which resource is more important than the others in this domain. For instance, if the purpose of the software agent is to ease people's lives rather than to complicate them, then maybe it should not be bothering people asking them about other people's email addresses.

Consider the machining domain [Min89], in which the task is to machine metal pieces into various shapes using a number of available machines such as drill-press, welding machine, and grinding machine. If we wanted to make a hole in a metal piece, there may be many courses of action we could follow: we could drill a hole in the object, or we could use a punching machine. The use of the drill machine may be more costly than the using the punching machine. Which plan should be preferred may depend on the costs of the machining operation(s) that the plan uses.

Clearly, to reason about plan quality, there must be a representation of plan quality knowledge. Value theoretic functions are one formalism that has been suggested by many researchers in AI planning and in operations research to represent plan quality knowledge [KR93, Wil96]. The AI planning and learning community has been slow to adopt such representations because of the commonly held belief that "domain independent planning is a hard combinatorial problem. Taking into account plan quality makes the task even more difficult" [AK97].

1.2 Problem Description

This thesis presents a framework that employs value theoretic functions to represent plan quality knowledge and uses this representation to automatically learn domain specific heuristics that allow a partial order planner to produce high quality plans. While plan quality is the main focus, planning efficiency is also an important concern. For instance, one obvious way of producing optimal quality plans is to let a planner search exhaustively, producing all possible plans, and then simply picking the best plan. Clearly, exhaustive search is extremely inefficient and impossible to do in a reasonable amount of time for most real-world problems. Thus the purpose of this work is to investigate if there is a way of learning and incorporating domain knowledge into partial order planners that allows them to efficiently produce high quality plans. More precisely the problem investigated in this thesis can be defined as:

Given

- a planning problem (in terms of an initial-state and goals)
- domain knowledge (in terms of a set of actions and plan quality knowledge that can be used to compute the quality of a *complete plan*[1])
- a partial-order planner

Find

- a set of domain specific rules that can be used by the given partial-order to produce higher quality plans, for similar problems in the future, than the plans that the given planner would have produced without learning these rules.

1.3 Contributions Of This Work

The main contribution of this work is the design, implementation, and evaluation of PIP (Performance Improving Planner), a planning and learning system that can automatically learn domain specific knowledge to improve the quality of plans produced by a partial-order planner. The framework and algorithms support two alternative approaches for improving plan quality. One is to acquire search-control rules that are used during the planning process. The second is to learn the so-called *rewrite rules* [AK97] that modify already complete plans to improve their quality. Both these approaches were empirically compared to evaluate their benefits and costs. The dependent measures were plan quality and planning efficiency. The results of the cross validation experiments performed on a number of benchmark planning domains (such as Transportation [UE98], Softbot [Wil96], and Process planning [Min89]) suggest that search control rules are more effective in improving both plan quality and planning efficiency than rewrite rules.

---

[1] I use the term *complete plan* throughout this document to refer to a plan that has all the actions needed to satisfy the goals. A totally ordered plan (i.e., a plan in which all the actions are ordered) must exist corresponding to a complete plan but a complete plan does not have to be a totally ordered plan.

Empirical experiments were also conducted to compare PIP with SCOPE [EM97], the only other planning and learning system that learns to improve plan quality for partial order planners. These results show that PIP's analytic techniques allow it to learn to improve plan quality with fewer examples. Finally, PIP was evaluated by systematically varying domain features to see how changes in various domain properties affect PIP's performance. The results show that PIP's learning techniques benefit most in the domains where:

- each problem has a number of solutions of different quality (i.e., plan quality matters),
- the system does not produce high quality solutions without any learning (i.e., there is something to learn), and
- the search trees of training and testing problems are *similar* (i.e., they share some subgoals).

1.4 Organization Of This Dissertation

In the next chapter, I provide a brief overview of AI planning research and various machine learning techniques that have been used to learn knowledge for improving planning efficiency and plan quality. This discussion provides the motivation for Chapter 3 which presents PIP's architecture and algorithms. PIP has a learning and a planning component. The knowledge learned by PIP's learning component can be stored either as search control rules or as rewrite rules. Chapter 3 only describes how search control rules can be learned and used. Chapter 4 describes how PIP's algorithms presented in Chapter 3 can be modified to design a system called PIP-rewrite that stores the learned knowledge as rewrite rules. PIP-rewrite uses the learned rewrite rules to produce higher quality plans for subsequent problems. Chapter 4 also includes an empirical comparison of PIP and PIP-rewrite. The results presented in Chapter 4 suggest that while both approaches lead to significant improvements in plan quality, using search control rules is a better strategy when both plan quality and planning efficiency are a concern. The first part of Chapter 5 presents results of empirical experiments done to compare PIP's performance with other planning and learning systems that learn to improve plan quality for partial order planners. The second part of Chapter 5 provides further evaluation of PIP using a number of artificial domains systematically designed to test PIP's performance along a number of dimensions. Chapter 6 provides some conclusions and discusses directions in which this work can be extended.

Chapter 2

Background

The choice of the learning technique and the type of the domain specific rules to be learned depends on the planning algorithm used. The first section of this chapter presents a brief overview of AI planning techniques. The following two sections review various machine learning techniques that can be used to learn domain knowledge to improve planning efficiency and plan quality.

2.1 The AI Planning Problem

The classical planning problem is defined as:

Given

- problem specification in terms of the initial state and goals and
- a set of actions

Find

- a sequence of actions that can transform the world from the initial state into a state where all the problem goals are true.

2.1.1 Knowledge Representation

Traditionally, planning problems are represented using the STRIPS language. In STRIPS, states are represented by conjunctions of propositional-attributes (represented by function-free ground literals). The propositions in a state are added or deleted by actions defined for a domain (and represented as *schemas*). Action schemas are represented by three components:

- *The action description:* The parameterized name for the action such as $fly(Plane, From, To)$ denotes the action of flying a plane $Plane$ from location $From$ to location To.

- *The preconditions:* A conjunction of propositional attributes that must hold for the action to work. For instance, the preconditions for the fly action could be that the plane has to be $at(Plane, From)$ in order for it to be flown from $From$ to To.

- *The effects:* A conjunction of propositional attributes that describes how the world changes by the application of the action. The effects are described by *add* and *delete* lists of propositional attributes made true and false (respectively) by the execution of the action. Propositions not mentioned in the effect set are assumed not to change by the application of the action. So for instance, we would want our fly action to add the effect $at(Plane, To)$ to the world-state, indicating that after flying the plane from $From$ to To, the plane is at location To. We would also want to indicate that the proposition $at(Plane, From)$ does not hold true after the execution of the fly action by encoding it as a delete-effect of the fly action.

STRIPS does not allow us to talk about metric resources such as time and money. This means that the problems such as the transportation problem defined in Figure 1.2 cannot be encoded into STRIPS. Veloso's Logistics domain [Vel94] is the transportation domain without any resources. Its encoding into STRIPS is shown in Figure 2.1.

STRIPS also does not allow conditional effects or universal statements in the effect set. These assumptions are too restrictive to represent most real-world planning domains. This has led to several extensions of STRIPS to allow more expressive constructs. Action Description Language (ADL) [Ped89] extends STRIPS to allow conditional effects, universal quantifications in preconditions and disjunctive preconditions. R-STRIPS [Wil96] extends STRIPS to allow it to represent and reason with resources. PIP uses a version

```
Action: load_truck(Obj,Truck,Loc),
       Preconditions: {at_obj(Obj, Loc),
                       at_truck(Truck, Loc)},
       Effects: {inside_truck(Obj, Truck),
                 not(at_obj(Obj,Loc))}

Action: unload_truck(Obj,Truck,Loc),
       Preconditions: {inside_truck(Obj,Truck),
                       at_truck(Truck,Loc)},
       Effects: {at_obj(Obj, Loc),
                 not(inside_truck(Obj, Truck))}

Action: drive_truck(Truck, Loc_from, Loc_to),
       Preconditions: {same_city(Loc_from, Loc_to),
                       at_truck(Truck, Loc_from)},
       Effects: {at_truck(Truck, Loc_to),
                 not(at_truck(Truck, Loc_from))}

Action: fly_airplane(Airplane, Loc_from, Loc_to),
       Preconditions: {airport(Loc_to),neq(Loc_from, Loc_to),
                       at_airplane(Airplane, Loc_from)},
       Effects: {at_airplane(Airplane, Loc_to),
                 not(at_airplane(Airplane, Loc_from))}

Action: unload_airplane(Obj, Airplane, Loc),
       Preconditions: {inside_airplane(Obj, Airplane),
                       at_airplane(Airplane, Loc)},
       Effects: {at_obj(Obj, Loc),
                 not(inside_airplane(Obj, Airplane))}

Action: load_airplane(Obj,Airplane,Loc),
       Preconditions: {at_obj(Obj, Loc),
                       at_airplane(Airplane, Loc)},
       Effects: {inside_airplane(Obj,Airplane),
                 not(at_obj(Obj,Loc))}
```

Figure 2.1: Veloso's logistics domain: a resource-less version of Transportation domain. In the Prolog tradition, capital letters are used to represent variables and small letters to represent constants here and elsewhere in this dissertation.

of R-STRIPS as its knowledge representation scheme which is presented in more detail in Chapter 3.

2.1.2 Search Techniques

Using the STRIPS language, the planning problem can be seen as a graph search problem. In a state-space search paradigm, the nodes of the graph are the possible states of the world and the arcs correspond to the legal moves that transform one state into another. The planning problem then is to find a path in this graph from a given initial state to a state where the given goals are true. Figure 2.2 shows the graph for the logistics problem of Figure 1.2. There are two actions that can be taken in the initial-state (Node 1) because their preconditions are true in that state: *load-truck(parcel, truck, edm-po)* and *drive-truck(truck1, edm-po, edm-ap)*. Taking each of these actions transforms the current state into a unique world state. So for instance, driving the truck to the Airport causes the truck to be at the Airport. The search algorithms that search through the space of states are known as ***state-space planners***.

State-space planners begin at one of the world states (typically, the initial-state) as their current state and proceed by applying an applicable action (i.e., an action whose preconditions are satisfied in the current state). With the application of each action, changes prescribed in the action's add and delete lists are applied to the current state to produce a new unique world state. By applying one action at a time, a state-space planner moves from the current state to an adjacent state in the state-space. It stops when it reaches a state where all its goals are satisfied. Using this representation, it is easy to see how various graph search strategies such as breadth-first-search or depth-first-search can be used to search for a solution. The worst case complexity of the problem is equal to the size of the graph. The size of the graph is exponential in the number of operators as well as the number of ways in which the operators can be instantiated. This two layered complexity places domain independent planning in the class of P-Space problems [ENS95].

The planning process can be a simple *forward search* from an initial state to a goal state, a *goal-directed search* that reasons backwards from the goals or

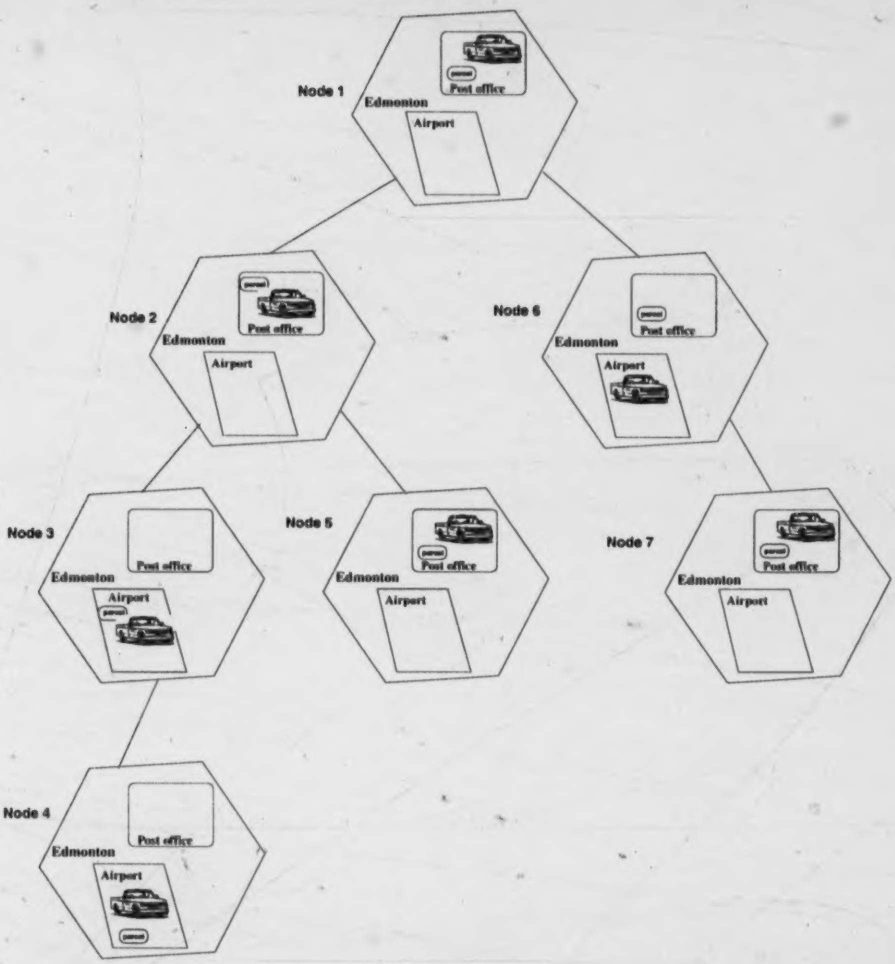

Node 1
Edmonton
Post office
Airport
parcel
Node 2
Edmonton
Post office
Airport
parcel
Node 6
Edmonton
Post office
Airport
parcel
Node 3
Edmonton
Post office
Airport
parcel
Node 5
Edmonton
Post office
Airport
parcel
Node 7
Edmonton
Post office
Airport
parcel
Node 4
Edmonton
Post office
Airport
parcel

Figure 2.2: Part of the state-space search tree for the Logistics problem of Figure 1.2.

a *bi-directional search.* Goal directed approaches are preferred in the domains in which a large number of actions can be executed in the initial-state but only a few actions can add the goals. Most planners use goal-directed search techniques such as means-ends analysis because they reduce the number of intermediate states in most interesting planning problems by focusing on those operators that can satisfy some outstanding goals.

Some early planning systems were built around the subgoal independence assumption (also called the *linearity* assumption) i.e., assumption that given a conjunct of goals, a plan can be formulated for each problem separately and then these plans can simply be concatenated to solve the conjunctive goal. This strategy works well if subgoals are independent but that is rarely the case even in simple toy problems such as many Blocksworld problems. As Sussman [Sus73] pointed out, interactions between goals (i.e., the solution for the second goal requiring undoing the first goal) cause most backtracks in the linear planning systems.

A variety of approaches have been suggested to deal with the goal interaction problem. The first such approach was adopted by STRIPS [FN71] itself. It involved solving the conjunctive goal, assuming that all subgoals are independent, and then concatenating the subplans to form a plan for the conjunctive goal. If the final plan is not a solution for the conjunctive goal (i.e., subgoals are not independent) then STRIPS attempts to solve the subgoals in a different order. This approach is very inefficient because it throws away the whole plan most of which may be correct (in fact, a correct plan may be just another ordering of the same actions). Sussman's HACKER [Sus73] and Tate's INTERPLAN [Tat74] improved this strategy by constructing a plan for a subgoal and then trying to extend it for the second goal. If the extension fails or undoes a previously satisfied goal, then these systems try to fix the problem by reordering the goals.

Initial-state: {at-object(package, edmonton-postoffice), at-truck(truck1, edmonton-postoffice), at-plane(plane1, edmonton-airport)}

Goal: {at-truck(truck1, edmonton-airport), at-object(package, edmonton-airport)}

Figure 2.3: A logistics problem. In the initial-state the object is at Edmonton Post Office and the goal is to get it to Edmonton Airport. There are two vehicles that can be used for transportation. Truck1 is at Edmonton Post Office and Plane1 is at Edmonton Airport.

A still more sophisticated way of dealing with goal interaction problem was suggested by Waldinger [Wal77]. If the planner needed to undo the first goal in order to achieve the second goal then Waldinger's planner backtracks and tries to make the second goal true at a different place in the plan for the first goal. For instance, suppose that the Waldinger's planner was trying to extend the plan *drive-truck(truck1, edmonton-postoffice, edmonton-airport)* to resolve the second goal *at-object(package1, edmonton-airport)* to solve the logistics problem shown in Figure 2.3. It would realize that it has to undo its first goal in order to achieve the second goal. Therefore, it tries to make the second goal true before the action *drive-truck(truck1, edmonton-postoffice, edmonton-airport)* and succeeds.

Waldinger's planner's main contribution was to decouple the planning order (i.e., the order in which actions are added during planning) from the plan order (i.e., the order in which actions are placed and executed in the final totally ordered plan[1]). But it limited this strategy to the goal-interaction cases. Sacerdoti's NOAH [Sac74] was the first to do this decoupling in general. This search strategy, now called *partial-order planning*, (and originally referred to as non-linear planning[2]) allows the actions to be unordered with respect to one another until some interaction is detected between them and only then ordering them. Unlike total order planners (such as STRIPS, Waldinger's planner, HACKER and INTERPLAN) that commit to a specific ordering of actions right away, partial-order planners (such as NOAH and Tate's NONLIN [Tat77]) leave a newly added action unordered with respect to the existing actions, ordering it only in response to future interactions.

A key aspect of partial-order planning is keeping track of past decisions and the reasons for those decisions. For example, if a planner adds an action to drive a truck from *edmonton-postoffice* to *edmonton-airport* to satisfy the goal of delivering a parcel to *edmonton-airport*, then the truck must be at *edmonton-postoffice*. If another goal causes it to move the truck elsewhere then it has to ensure that the truck comes back before its previously added action *drive-truck(truck1, edmonton-postoffice, edmonton-airport)* can be executed. A good way of ensuring that the different actions introduced for different goals do not interfere is to record the dependencies between actions explicitly. To record these dependencies, NONLIN invented the data structure called a *causal link*.

---

[1] From here on, we refer to the final totally order plan as simply *the final plan*.

[2] Most recent authors prefer the term partial-order planning and reserve the term non-linear planning for those planning systems that allow interleaving of goals and do not solve the goals in a strict order.

If an action p adds a proposition e to satisfy a precondition of an action c then $p \xrightarrow{e} c$ denotes the causal link.

Most AI researchers considered partial-order planning as more efficient than total-order state-space planning since premature ordering commitments are delayed until more informative decisions can be made. However, these intuitions were not put to empirical test until a conceptually simpler and more accessible version of NONLIN, now called SNLP, was presented by [MR91]. Unlike NONLIN, SNLP is also systematic in that a planning node is guaranteed to be visited only once. SNLP uses *causal links* to record the purpose for introducing a step into a plan and to protect that purpose. SNLP's key innovation is a methodical technique for creating and protecting causal links. SNLP labels a causal link $p \xrightarrow{e} c$ as *threatened* if some step t may possibly be ordered between p and c such that it deletes a precondition that matches e. SNLP protects a causal link by promoting t to come before p (i.e., adding an ordering constraint $t \prec p$) or by demoting t to come after c (i.e., adding an ordering constraint $c \prec t$).

Barrett and Weld [BW94] compared SNLP with various state space planners and showed that it significantly outperforms total-order planning algorithms on a number of different domains including domains with independent subgoals, interacting subgoals, and complex operator-selection decisions.

The crucial point is that partial-order planning can be seen as a refinement process, i.e., a process of progressively adding more constraints to a partial plan until all its flaws are removed and a complete plan is obtained. Each planning decision to ***add-action*** or ***establish*** can be seen as adding a causal-link constraint and an ordering constraint while a ***promotion/demotion*** decision can be seen as only adding an ordering a constraint.

2.1.3 Decision Theoretic Planning

One reason that domain independent planners have had limited success in being applied to real world problems is their inability to produce high quality plans. Decision theory provides a method for choosing among alternatives and provides a language that allows reasoning about plan quality. Decision theoretic planners aim at combining the domain independent AI planning techniques with utility models from decision theory to represent and reason with plan quality.

Although decision theory provides a method for choosing among alternative plans, it provides no guidance in structuring planning knowledge, no way of generating alternatives, and no computational model for solving planning problems [HH98]. A naive approach to generating optimal quality plans would be to generate all viable plans using an AI planner, compute their quality values using the decision-theoretic quality function, and return the best quality plan. This approach is clearly inefficient and impractical for most real world planning situations. To plan effectively, the planner must be able to evaluate the potential quality of a partial plan and to pursue higher quality alternatives. The approach followed by [FS75] was to add restrictions (probability and utility models) to classical planning algorithms. DRIPS [HH94] structures actions into an abstraction hierarchy and focuses on partial plans whose utility is computed to be within an interval. PYRRHUS [Wil96] extends DRIPS techniques to partial-order planning. It uses branch and bound search in the space of partial plans. Each time a complete plan is generated, PYRRHUS computes its exact quality value and compares it to the best so far. If it is better, it is kept and the bound updated. Partial plans with a lower quality value are discarded and planning terminates when no partial plan is left to be refined. Because of its exhaustive search PYRRHUS is guaranteed to find an optimal quality plan, given enough computational resources. Williamson [Wil96] reports that adding hand-coded heuristics to PYRRHUS allows it to find optimal quality plans more efficiently. However, manually encoding search control rules is time consuming and difficult because it requires in-depth knowledge of both the planning algorithm as well as the domain. Automatically acquiring such heuristics is a more challenging problem.

2.2 Learning to Improve Planning Efficiency

Coupling a domain independent planning system with a machine learning engine to allow it to *automatically* acquire domain specific knowledge to limit the search for a viable plan has a long history in AI. STRIPS [FN71] itself used the triangle-table analysis to learn from its own experience. However, not all planning and learning systems learn from their own experience. Some systems (called *apprenticeship learning systems*) can also learn by interacting with their users [MMST93]. These systems aim for partial automation of the knowledge acquisition process which is still useful because it promises to eliminate the role of the knowledge engineer.

Various machine learning techniques have been used to learn domain-specific heuristics to improve planning efficiency for both state-space as well as partial-order planners. The focus of these speed-up learning techniques is to learn the association between a search-state and a planning decision so as to prevent the planner from taking as many bad planning decisions (the ones that lead it to dead-ends and have to be backtracked from) as possible and to do so as early as possible (during the search). These associations are called *search control knowledge* because they are used by planning algorithms to limit the search to those branches that have in the past led to planning success. The rest of this chapter reviews these machine learning techniques used to acquire search control knowledge for planners.

The machine learning techniques are organized along the dimensions of *inductive*, *analytic* and *case-based* learning techniques. Analytic learning systems use proof procedures and some representation of semantic domain knowledge and perform extensive analysis of a single example to modify their knowledge base about the domain. This is in contrast to the inductive approaches that typically perform syntactic comparisons between feature value vectors of a large number of examples to identify the features that empirically distinguish the positive examples of a concept from the negative examples. In contrast to the methods that construct an explicit representation of the target function, case-based approaches simply store the training examples.

2.2.1 Inductive Learning Techniques

Initial-state: {at-object(obj, edm-po), at-truck(tr1, edm-po), at-plane(pl1, edm-ap), at-truck(tr2, cal-ap), same-city(edm-po, edm-ap), same-city(cal-po, cal-ap)}

Goal: {at-object(obj, cal-ap)}

Figure 2.4: A planning example from Veloso's logistics domain. In the initial-state the object is at Edmonton post office (*edm-po*) and the goal is to get it to Calgary Airport (*cal-ap*).

Inductive learning techniques, also known as *similarity based learning* techniques, acquire search control rules by comparing training examples with one another to find features that empirically distinguish positive from negative training examples. The training examples can be found by solving a set of training problems and labeling the planning decisions on a path that leads to a successful plan as positive examples and the planning decisions on a path that leads to a dead-end as negative examples. Consider the example from Veloso's logistics domain shown in Figure 2.4. In this example, the goal is to deliver an object (*obj*) from Edmonton Post-office (*edm-po*) to Calgary Airport (*cal-ap*). Figure 2.5 shows part of the search-tree for this problem. There are two possible plan-refinements that can be applied to the partial plan in Node 1: *add-action: unload-truck* or *add-action: unload-plane*. Since this is an intercity delivery and in Veloso's logistics domain only planes can be used to fly objects between cities, application of *add-action: unload-truck* will lead to failure. Hence Node 2 will be labeled as a negative example of the application of the refinement *add-action: unload-truck*. Node 10, on the other hand, is a positive example of the application of the refinement *add-action: unload-truck* because it leads to success. The inductive learning task then is to search through the space of all possible rules (limited by the language of the learner) to select a rule that covers all the positive examples and none of the negative ones.

However, the search space can become very large very quickly. Therefore, inductive systems need to limit their language. For instance, SCOPE [EM97], a system that uses Inductive Logic Programming (ILP) [MR94] to learn search control rules for partial-order planners, limits the search to the literals present in the proof-trees of all the planning problems from which negative and positive examples were drawn as well as some predefined combinations of these literals. Figure 2.6 shows the control rule learned by SCOPE for the application of the plan refinement ***add-action: unload-truck***.

2.2.2 EBL

Explanation Based Learning (EBL) [MKK86, Min89] is an analytic technique that can be used to learn from a single example of planning success or failure. Unlike inductive learning techniques, explanation-based learning systems use domain knowledge to explain each training example to infer the example features that are relevant to its planning success/failure. Given a problem, the planner searches through the search-space and returns a solution. The learner then explains the failures and successes in the search tree explored by the planner and uses these explanations to generate search control rules that may be used by the planner to avoid the failing paths and bias it towards the previously successful paths.

In SNLP+EBL [KKQ96], a system that uses EBL to learn search control rules for SNLP, learning is initiated whenever the planner detects a planning failure. The system detects that a search node N' is a dead end when every possible refinement (i.e., constraint addition) leads to inconsistency with some previously added constraint. The set of inconsistent constraints in the partial plan forms the failure explanation at the dead end node. This explanation is then regressed backwards (in the search tree) to a higher level node N where some unexplored paths remain. The explanation at a higher node is the conjunction of the failure explanations of all its children and the flaw at that node. The purpose of regressing the failure explanation backwards is to determine the minimal set of constraints that must be present at N such that after taking the decisions $d_N, ..., d_{N'}$ the system ends up adding inconsistent constraints and reaches a dead-end node N'. This explanation then is generalized and converted into search control rules that can be used to

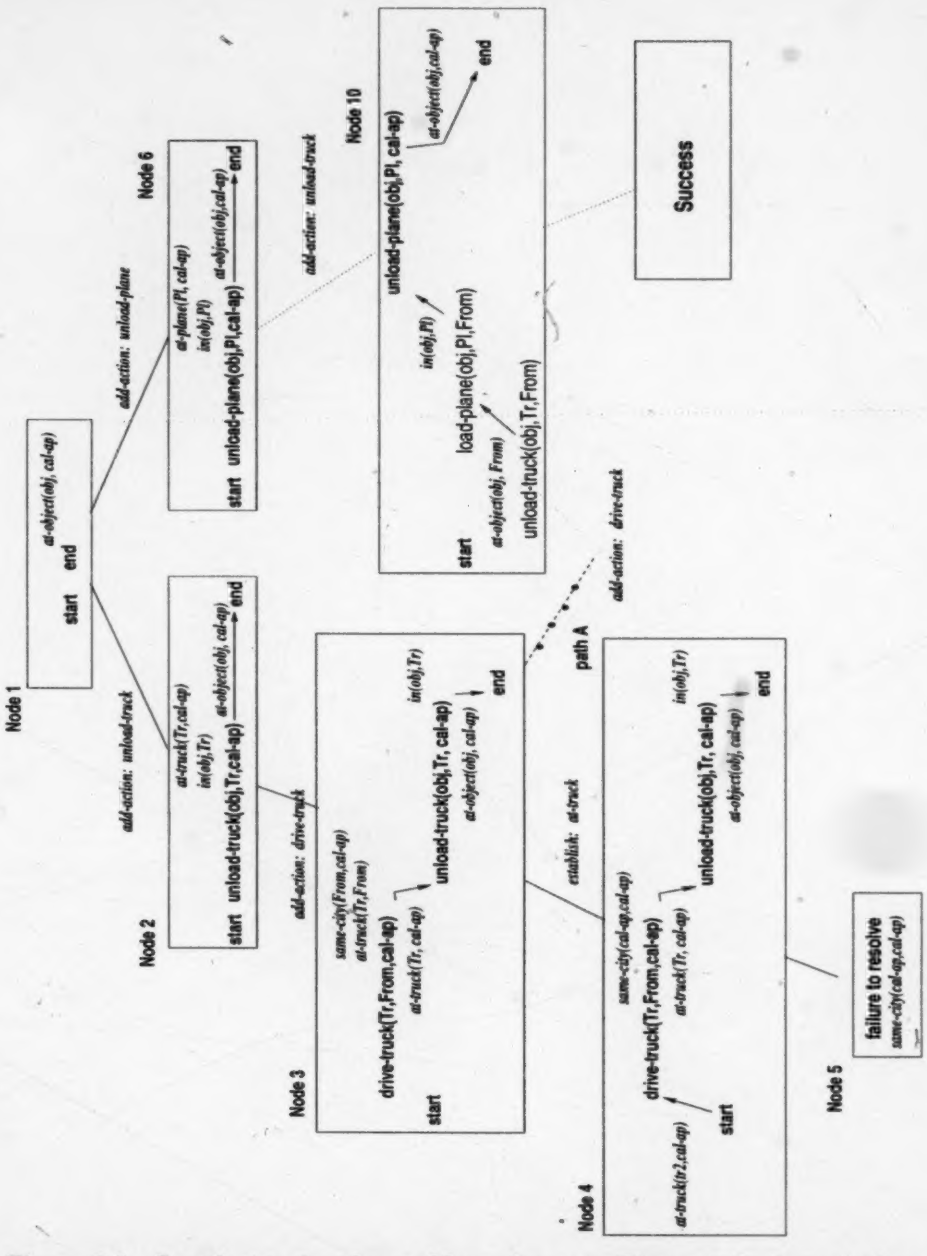

Node 1
start end
at-object(obj, cal-ap)
add-action: unload-truck
add-action: unload-plane
Node 2
start unload-truck(obj,Tr,cal-ap) end
at-truck(Tr,cal-ap)
in(obj,Tr)
at-object(obj, cal-ap)
Node 6
start unload-plane(obj,Pl,cal-ap) end
at-plane(Pl, cal-ap)
in(obj,Pl)
at-object(obj,cal-ap)
add-action: unload-truck
Node 10
start
at-object(obj, From)
unload-truck(obj,Tr,From)
load-plane(obj,Pl,From)
in(obj,Pl)
unload-plane(obj,Pl, cal-ap)
at-object(obj,cal-ap)
end
Success
add-action: drive-truck
Node 3
start
drive-truck(Tr,From,cal-ap)
same-city(From,cal-ap)
at-truck(Tr,From)
at-truck(Tr, cal-ap)
unload-truck(obj,Tr, cal-ap)
in(obj,Tr)
at-object(obj, cal-ap)
end
path A
add-action: drive-truck
establish: at-truck
Node 4
start
drive-truck(Tr,From,cal-ap)
same-city(cal-ap,cal-ap)
at-truck(tr2,cal-ap)
at-truck(Tr, cal-ap)
unload-truck(obj,Tr, cal-ap)
in(obj,Tr)
at-object(obj, cal-ap)
end
Node 5
failure to resolve
same-city(cal-ap,cal-ap)

Figure 2.5: Search-tree for the problem shown in Figure 2.4. I use italics to represent open preconditions which are treated as subgoals. When these preconditions are still open (i.e., have not been satisfied), they are displayed next to the action that requires them. Arrows between actions denote causal-links showing which subgoals of an action have been satisfied. The arrow direction is from producer to the consumer of a condition.

if open-condition(at-object(Object, To), A1) & member(effect(at-object(Object, Loc), A2), P)
& member(same-city(To, Loc), Init-state) & consistent(order(A2, A1), P)
then
apply decision {add-action unload-truck(Object, From, To) to partial plan P}

Figure 2.6: Search-control rule learned by SCOPE

avoid similar failures.

To see how EBL can be used in the context of a partial-order planner, consider again the logistics problem introduced in Figure 2.4. A partial-order planner such as SNLP recognizes that Node 5 is a dead-end because it cannot find any action, existing or new, that adds the condition *same-city(cal-ap, cal-ap)*. Hence the reason for the failure of this node is that the effect *same-city(cal-ap,cal-ap)* was not present in the initial-state[3]. This explanation can then be regressed backwards to its parent node (Node 4). Since the failure constraint is not added by Node 4, regressing it over this node does not change the explanation. At Node 3 another path (path A in Figure 2.5) is available, hence the failure explanation is not regressed any further. It is simply generalized and stored in the form of the search-control rule shown in Figure 2.7.

If open-condition(at-truck(Tr, F)) & effect(at-object(Object, Loc), A)
& not(member(same-city(F, T), Init-state)
then do not apply-decision {establish(at-truck(Tr, F))}

Figure 2.7: Search-control rule learned by SNLP+EBL

When the planner is solving a similar problem whose initial state does not contain *same-city(F, T)* and it generates partial plan that contains the open condition *at-truck(Tr, F)* and effect *at-object(Object, Loc)* then the search control rule tells the planner not to try the decision *establish: at-truck(Tr, F)* because it will lead to a dead-end. Thus the search control-control rules learned by SNLP+EBL help it avoid the dead-ends and lead it to improving its planning efficiency.

---

[3] Since no action adds this condition as an effect, the only way this condition can be satisfied is if it is present in the initial-state.

2.2.3 Case-based Learning

Learning is an essential part of Case-Based Reasoning (CBR) [Kol93]. Case based reasoning means remembering successes so that they can be reused, remembering failures so that they can be avoided, and remembering repairs to solutions so that they can be reapplied [Ham90]. Planning systems that use CBR can be divided into two categories: ***plan reuse*** [Ham90] and ***derivational replay*** [Ihr96]. The plan reuse systems such as CHEF [Ham90] store the final plan which is the product of the planning episode. The derivation replay systems such as PRODIGY/ANALOGY [Vel94], on the other hand, store the ***planning decisions*** made during planning. For instance, the case stored by a plan-reuse system for the example shown in Figure 2.4 contains the final plan:

```
load-truck(Object, Truck1, City1_po)
drive-truck(Truck1, City1-po, City1-ap)
unload-truck(Object, Truck1, City1_ap)
load-plane(Object, Plane1, City1_ap)
fly-plane(Plane1, City1-ap, City2-ap)
unload-plane(Object, Plane1, City2-ap),
```

and the case stored by a derivational replay system (such as DerSNLP) consists of the planning decisions:

```
1- add-step unload-plane(Object, P, City2-ap),
2- add-step load-plane(Object, P, C),
3- add-step fly-plane(P, C, City2-ap),
4- establish precond at-plane(P, C) of load-plane with
                              at-plane(Plane1, City1-ap)
5- establish precond at-plane(P, C) of fly-plane with
                              at-plane(Plane1, City1-ap)
6- add-step unload-truck(T, City1-ap)
7- add-step load-truck(T, C2)
8- add-step drive-truck(T, C2, city1-ap)
9- establish precond at-truck(T, C2) of load-truck with
                               at-truck(Truck1, City1-po)
10- establish precond at-truck(T, C2) of drive-truck with
                               at-truck(Truck1, City1-po)
11- establish precond at-object(Object, C2) of load-truck with
                               at-object(Object, City1-po)
```

Storing planning decisions is more useful in the situations where planning decisions are applicable in a more general set of situations than the final plan.

Case-based learners remember solutions or fragments of solutions. In order to use them in future, these solutions must be indexed in ways that allow the planner to recognize that they are relevant to the current problem. The basic idea is the same as EBL, namely, to bias the planner to take the paths that have led to success in the past for similar problems. Case-based planning (specially derivational replay systems) can be considered as performing analytical learning from planning successes. Given a problem shown in Figure 2.4, a derivational analogy system (such as PRODIGY/ANALOGY and DerSNLP) learns the *essential* features in the initial state whose presence guarantees that the stored planning decisions will be applicable to a new problem having those features. For instance, the above case is indexed by the following *relevant initial conditions*[4] and the problem goal by DerSNLP.

```
at-object(Object, City1-po)            required by decision 11
at-truck(Truck1, City1-po)             required by decisions 9 & 10
at-plane(Plane1, City1-ap)             required by decisions 4 & 5
```

2.2.4 Hybrid Techniques

The complementary nature of various learning techniques suggests that planning and learning systems can be built to use more than one technique. For instance, derivational replay and EBL from failure have been combined in the system DerSNLP+EBL that learns from planning successes as well as from failures. Ihrig and Kambhampati [IK97] show that combining these two techniques yields significantly better performance improvements in a number of planning domains over using either technique alone.

EBL and inductive learning techniques can also be combined in various ways to take advantage of the strengths and weaknesses of each. Most of the EBL+inductive systems first learn search control rules using EBL and then generalize those rules. This strategy was first used by LEX-2 [Mit83] and MetaLEX [Kel87]. Other systems, such as *lazy explanation-based learning* systems [BV94], do not build complete explanation proofs and instead generate incomplete explanations and then incrementally refine them using subsequent examples. This is the strategy used by HAMLET [BV94] to learn search control rules for PRODIGY. HAMLET generates a bounded explanation of each planning decision and stores it as a rule. These rules are specialized if they are found to cover negative examples and generalized if they exclude a positive example.

---

[4] These are the initial conditions that satisfy the preconditions of a planning decision that is being stored in the case.

2.3 Learning to Improve Plan Quality

Considerable planning and learning research has focussed on the problem of learning domain knowledge to improve planning efficiency. Less attention has been paid to the problem of learning domain knowledge to improve plan quality. The reason being that most speed up learning systems (like most planning systems) define planning success rather narrowly, namely as the production of any plan that satisfies the goals. As planning systems are applied to real world problems, concern for plan quality becomes crucial. Many researchers have pointed out that generating good quality plans is essential if planning systems are to be applied to practical problems [Wil88, RK93, DGT95, Per96].

2.3.1 Plan Quality Measurement Knowledge

The main reason why it is difficult to extend the existing planning and learning systems to deal with plan quality is that these systems do not possess plan quality measurement knowledge. This means that they cannot recognize the learning opportunities because they cannot express or evaluate plan quality as a concept. Speed up learning systems can generate multiple learning opportunities by simply computing the first viable plan. The positive examples are the search nodes on a successful planning path and negative examples are the search nodes on a failed path. In the plan quality context, a positive example would be a node that leads to a better quality plan and a negative example a node that leads to a lower quality plan.

The crucial point is that betterness/worseness of a planning path is relative with respect to another planning path, whereas success/failure of a search path is not. This means that unlike speed-up learning systems whose learning opportunities consist of a single path, a learning opportunity for quality rule learning systems must consist of at least two planning paths that lead to plans of different quality. Such learning opportunities cannot be generated by following the classical planning approach of stopping after constructing the first plan. Instead, the learning algorithm needs at least two qualitatively different planning paths to learn something. These two plans can be generated by computing the first plan and then backtracking to a choice point where an unexplored path remains and exploring it to compute an alternative plan, or by running the planner using two different search strategies such as depth-first search and bread-first search to produce two different plans, or by asking a user to provide an alternative plan for the same problem (like an apprenticeship system).

Given a problem, the so-called apprenticeship learning systems produce a solution for the problem and then ask the user for a better quality solution [MMST93]. The learning can then proceed by comparing the better quality solution with the worse quality solution. However, such systems demand too much from their users. The user (or users) must consistently provide the learning system with better quality solutions. If the user is not consistent and ever presents the system with a lower quality solution, the system can easily be misled into learning the wrong information. This limits the type of situations in which systems that do not represent quality knowledge can be applied.

A learning system can only identify learning opportunities without a user's help if it possesses the knowledge required to measure the quality of a plan. The term plan quality has been used in the AI planning literature to refer to a variety of concepts. Some of these include:

- plan length,
- resources consumed by execution of the plan,
- robustness of the plan.

Of the planning and learning systems that do consider plan quality as a criterion, most use plan length as a measure of plan quality. This metric can suffice, if each possible action has the same unit cost and there is no sense in which the plan's execution uses or impacts other domain resources. However, it is clearly insufficient in the types of domains discussed in the first chapter. In such domains, plan quality is a function of the resources consumed by the plan. In order to measure the quality of a plan in such domains, the necessary knowledge is:

1. amounts of the resources in the initial state,
2. amounts of resources each action consumes, and
3. the relative importance of each resource in the domain.

It is this knowledge that I call plan quality measurement knowledge (or simply plan quality knowledge).

The systems that possess plan quality knowledge can generate their own learning opportunities by generating two alternative plans, evaluating their qualities, and learning if their quality values are different. Such systems can also learn in the apprenticeship mode. When learning from a user, a system possessing plan quality knowledge has the flexibility of rejecting a user's advice if the user ever presents it with a lower quality plan or it can learn how *not* to plan.

Analytical learning systems (such as PRODIGY/EBL and SNLP+EBL) use axioms to explain the failure or success of a search node. For instance, following are some of the axioms used by SNLP+EBL to construct its explanations:

1. a search node is a failure node if it has two inconsistent constraints.
2. a search node is a failure node if all its children fail.
3. in Blocksworld problems, no block can have another block on top of it and be clear at the same time.

The first two axioms are domain independent and can be applied in any domain whereas the third axiom represents information that is specific to the Blocksworld domain. Axioms such as these can explain a planning failure. However, they are of no help in explaining why a search path leads to a better/worse quality plan. Betterness/worseness of a search node must be explained by appealing to the plan quality knowledge.

2.3.2 Analytic Techniques for Learning to Improve Plan Quality

Most of the work on using analytic techniques to learn search control rules to improve plan quality has been done to learn search control rules for the state-space planner, PRODIGY [VCP+95].

QUALITY [Per96] is a learning system that uses an analytical approach to learn control rules for PRODIGY. Given a problem, a quality metric and a better quality plan, QUALITY assigns a cost value to the nodes in the better quality plan's trace and to the nodes in the system's default planning trace. It identifies all those goal-nodes that have a zero cost in the better plan's trace and a non-zero cost in the default trace. Assuming that all the actions are assigned a positive cost value, cost of achieving a goal can only be zero either because it is true in the initial state or because it was added as a side-effect of an operator added to achieve another goal. The reason for the difference in the cost values of the two nodes is identified by examining both search trees. The explanation thus constructed forms the antecedent of the control rule learned by QUALITY. This algorithm limits QUALITY to learn search control rules from only those decision points where where one branch has a zero cost and the other a non-zero cost.

Iwamoto [Iwa94] also reports on an analytic learning algorithm to learn search control rules for PRODIGY to find near-optimal solutions in LSI design. Unlike QUALITY, the quality is explicitly represented as part of the goal. The goals consist of two parts, necessity goals and quality goals. Necessity goals are the propositional predicates used in STRIPS style planners. The quality goals specify the minimally acceptable quality of the plan. For instance, in LSI design a quality goal is "find a circuit where total cell number is less than or equal to 4." The planner then exhaustively searches until it finds a plan that satisfies the necessity goals as well as the quality goals. If PRODIGY had to construct more than one solution to come up with the first acceptable plan, then two different quality solution paths are compared to explain the reason for the betterness/worseness of a planning path. The explanation is constructed by back-propagating the weakest conditions, but excluding the conditions expressed in terms of the predicates related to quality. Iwamoto's technique only learns from differences in *add-action* planning decisions and does not take advantage of other learning opportunities.

2.3.3 Non-analytic Techniques for Learning to Improve Plan Quality

Since inductive learning techniques do not *explain* the success or failure of a search node, they do not need the plan quality knowledge to learn quality improving rules as long as the learning opportunities are identified by a user. Inductive techniques such as SCOPE [EM97] can be used to learn plan quality improving rules without any changes to the learning algorithm itself. Given a planning problem to solve and an optimal plan for that problem, SCOPE considers each of user's refinement decisions to be a positive example of the application of that refinement and the system's refinement decision to be a negative example. These positive and negative examples are then passed to an inductive concept learner to induce a rule that covers all positive examples and none of the negative examples.

Instead of comparing just two qualitatively different planning episodes, HAMLET [BV94] explores the space of all possible plans to find the optimal plan(s). It does not use the quality function to construct an explanation of the betterness/worseness of a planning decision. Instead, HAMLET learns rules saying that the choices made by the optimal path should be preferred over other available choices whenever the planner is at that node during the search. These rules are then compared to previously learned rules with a view to generalize. The generalized rules are then stored. If a general rule leads to a non-optimal plan then it is specialized and the general rule is forgotten.

2.3.4 Planning by Rewriting

Most previous research on learning to improve plan quality has focussed on extending the speed-up learning techniques to learn search control rules that can bias the planner towards making the choices that lead it to produce better quality plans. An alternative technique for improving plan quality is *planning by rewriting* [AK97]. Under this approach, a planner generates an initial (possibly lower quality) plan, and then a set of rewrite rules are used to transform this plan into a higher-quality plan. Unlike the search control rules for partial-order planners (such as those learned by SNLP+EBL [KKQ96] and SCOPE [EM97]) that are defined on the space of partial plans, rewrite rules are defined on the space of complete plans.

Plan rewriting is related to graph, term and program rewriting [BN98]. A rewrite system is specified as a set of rules that encode the equivalence relationship between two terms/graphs/programs. When extending this approach to planning, two subplans are considered equivalent if they solve the same problem. Figure 2.8(a) shows an example of a plan-rewrite rule. This rule denotes that the subplans {*load-truck(Object, Truck, Loc), unload-truck(Object, Truck, Loc)*} and {} are equivalent i.e., loading and unloading an object at the same place is equivalent to doing nothing. Such knowledge can be used to delete the actions *load-truck(Object, Truck, Loc), unload-truck(Object, Truck, Loc)* from any plan that contains them, presumably improving the quality of the plan.

It has been argued that plan-rewrite rules are easier to state than search control rules, because they do not require any knowledge of the inner workings of the planning algorithm [AK97]. That may partially explain why most of the search-control systems have been designed to automatically learn search-control rules, whereas the only existing planning by rewriting system, Pbr [AK97], uses manually generated rewrite-rules. Pbr used a small number of hand-coded rewrite rules to improve the quality of the plans produced by SAGE [Kno96], a partial-order planner for Blocksworld [AHT90], a process planning domain [Min89] and a query planning domain [AK98].

```
replace
        actions: {load-truck(Object, Truck, Loc), unload-truck(Object, Truck, Loc)}

with
        actions: {}
```

(a) A rewrite rule

```
If      open-condition(at-object(Object, Loc))  &  effect(at-object(Object, Loc), A)

then    apply-decision {establish(at-object(Object, Loc)) with A's effect}
```

(b) A search-control rule

Figure 2.8: A search control and a rewrite rule learned from the same opportunity

One benefit of planning by rewriting is that the planning module itself does not have to be modified. This also means that any speed-up learning system can be used to efficiently produce an initial plan[5] which can be transformed into a higher quality plan using the rewrite rules. Another benefit is that, unlike search control rules, rewrite rules operate on complete plans and hence are easier to understand and debug. This is important if humans are involved in the planning loop (which is invariably the case in most critical applications).

Initial-state: {know-email(jonn), know-name(jonn), has-plan-file(jonn)}

Goals: {know-address(jonn) , know-phone(jonn)}

System's Plan	Model Plan
hire-cyber-detective(jonn)	**finger(jonn)**

Figure 2.9: A Softbot planning problem and two solutions for it.

The task of a rewrite rule learner is to identify two sequences of actions that are equivalent in their final effects: a *to-be-replaced* action sequence and

---

[5] Indeed, any state of the art planner such as Blackbox [KS99] can be used to generate the initial plan. This issue is further discussed in Section 6.2.5.

Initial-state: {at-object(letter1,edm-po), at-object(letter2, edm-ap), at-plane(plane2, edm-ap), at-truck(truck1, edm-po), at-plane(plane1, edm-ap)}

Goals: {at-object(letter1, cal-ap), at-object(letter2, cal-ap)}

System's Default Plan	Model Plan
load-truck(letter1, truck1, edm-po)	load-truck(letter2, plane2, edm-ap)
drive-truck(truck1, edm-po, edm-ap)	load-truck(letter1, truck1, edm-po)
unload-truck(letter1, truck1, edm-ap)	drive-truck(truck1, edm-po, edm-ap)
load-plane(letter2, plane1, edm-ap)	unload-truck(letter1, truck1, edm-ap)
fly-plane(plane1, edm-ap, cal-ap)	load-plane(letter1, plane2, edm-ap)
unload-plane(letter1, plane1, cal-ap)	fly-plane(plane2, edm-ap, edson-ap)
unload-plane(letter2, plane1, cal-ap)	fly-plane(plane2, edson-ap, cal-ap)
	unload-plane(letter1, plane2, cal-ap)
	unload-plane(letter2, plane2, cal-ap)

Figure 2.10: A Transportation planning problem and two solutions for it. The model plan is longer (i.e., has a larger number of steps) than the system's default plan but it is preferred because it consumes fewer resources.

a sequence of *replacing* actions. At first glance, it may seem that the rewrite rules can be learned simply by performing a syntactic comparison of the two complete plans. For instance, consider the case of two trivial plans shown in Figure 2.9. It is easy to see that *hire-cyber-detective* is the action to be replaced and *finger* is the replacing action. However, in case of anything more complicated than this trivial example, it is not possible to compute the local replacing and to-be-replaced actions by comparing the complete plans. For instance, consider the scenario from Veloso's logistics domain shown in Figure 2.10. It is easy to learn a global rewrite rule saying the system's plan can be replaced by the model plan. However, if we wanted to learn a local rule (which may be more general than the global rule) then we would have to compare the causal structure of the two plans. For instance, PIP-rewrite, the planning and learning system presented in Chapter 4, learns the local rule shown in Figure 2.11 by comparing the causal-link constraints associated with the system's plan and the model plan of Figure 2.10. The learning task for both a rewrite and a search-control learner then is (a) to analyze how two different constraint-sets that were added by the two different planning episodes lead to differences in

```
replace:
    actions: {fly-plane(Plane, City1-ap, City2-ap), fly-plane(Plane, City2-ap, City3-ap)}
with:
    actions: {fly-plane(Plane, City1-ap, City3-ap)}
```

Figure 2.11: Part of the rewrite rule learned by PIP-rewrite from the training problem shown in Figure 2.10

overall quality and (b) store that analysis in a form that is usable to produce better quality plans for similar problems.

2.4 Summary

Several domain independent methods have been developed for finding a plan for a given planning problem. Planners that use the least commitment strategy of partial-order planning are known to be more efficient than older state-space planning methods. However, the performance of even the most efficient domain independent planners is insufficient for real world problems. There is considerable evidence that incorporating domain specific heuristics into the domain independent planners can improve their planning efficiency and plan quality. However, manually encoding these heuristics is very expensive. Machine learning for planning offers a possible solution by automatically learning domain specific heuristics for planners. Most of this work has focussed on learning rules to improve planning efficiency and less work has been done to learn to improve plan quality. Various learning techniques such as inductive and analytic techniques have been applied for this purpose. The more powerful analytic techniques require more knowledge but can learn using a few examples, whereas inductive techniques do not require any background knowledge but need a large number of training examples to learn. It appears very difficult to use analytic learning techniques to learn quality improving domain specific search control rules for partial-order planners because so little information is available during partial-order planning.

An alternative approach for improving plan quality has been recently suggested. It involves efficiently producing a low quality initial plan and then modifying it using domain specific rewrite rules to turn it into a high quality plan. Automatically learning plan rewrite rules is a challenging problem that has not been addressed by previous researchers. Since the focus of this work was on exploring various techniques for learning to improve plan quality for partial-order planners, I was interested in investigating if plan-rewrite rules can be automatically learned and how they compare to the search control rules. The rest of this dissertation presents an analytic learning technique called PIP for learning search control as well as rewrite rules to improve plan quality of the plans produced by partial order planners.

Chapter 3

The PIP Framework

> 'This,' said Mr Pumblechook, 'is Pip.'
>
> 'This is Pip, is it?' returned the young lady, who was very pretty and seemed very proud; 'come in, Pip.' (page [Dic73])

This chapter introduces core ideas of the PIP framework. The first section presents PIP's knowledge representation scheme followed by PIP's architecture and algorithms.

3.1 Knowledge Representation Scheme

3.1.1 Value Functions for Quality

It has been widely acknowledged in both the theoretical and practical planning camps that plan-quality for most real-world problems depends on a number of (possibly competing) factors [KR93, Wil96]. I agree with Keeney and Raiffa [KR93] that most interesting planning problems are multiobjective.

The assumption underlying this work is that complex quality trade offs can be mapped to a quantitative statement. There is a long history of methodological work in operations research that guarantees that a set of quality-tradeoffs (of the form "prefer to maximize X rather than minimize Y") can be encoded into a value function, as long as certain rationality criteria are met [Fis70, KR93]. Value-theoretic functions are a well-developed mechanism devised by operations research workers to represent the evaluation function for multiobjective problems. A value function is defined on the outcome (i.e., the final-state) of a complete plan.

The first task towards the formulation of a value function is identification of the decision attributes. Keeney and Raiffa [KR93] suggest a hierarchical refinement scheme starting with the ***highest level objectives*** and refining them down to the ***low level measurable*** attributes. For instance, the overall objective of an agent using a transportation system may be "to have a good trip" which can be refined down to the measurable attributes such as "minimize door-to-door travel time" and "minimize fare costs." Once various objectives have been identified, the next step is to elicit the user's degree of preference of one attribute over another. Operations research and choice modeling researchers study different techniques for eliciting domain expert's preference knowledge [Hen81, dH90]. Based on the expert's responses to various combinations of multiple decision attributes, techniques such as conjoint analysis [Lou88] are used to estimate attribute utilities and to encode the revealed preference structure into a value function V.

$$V : D \times D \rightarrow \Re$$

where D is the set of decision attributes and $\Re$ is the set of real numbers.

If an agent's preferences constitute a partial-order over outcomes and satisfy certain rationality criteria (such as transitivity), the central theorem of decision theory [Fis70] states that these preferences can be represented by a real-valued ***value function*** V such that if s_1 and s_2 denote two outcomes then s_1 is preferable to s_2 i.e., $s_1 \succ s_2$ iff $V(s_1) > V(s_2)$. Even if the agent's preferences do not form a partial-order, the value function can still be used to form good approximations [Yu85]. Many AI planning researchers [FS75, Wel93, Wil96, HH98] have indeed used value functions to solve AI planning and reasoning tasks.

3.1.2 Representing and Reasoning with Resources

We assume that a value function defined on the resource levels for a domain is supplied to PIP along with the rest of the action definitions for the domain.

PIP uses a modified version of R-STRIPS[1] [Wil96], called PR-STRIPS, that allows it to represent resource attributes and the effects of actions on those resources. The basic idea is to deal with resources in an action centered manner i.e., each action specifies how it affects the resources. Numerical quantities of resources are denoted by *metric attributes*. Metric attributes are essentially treated like propositional attributes in the way they enter the state description and an action's preconditions and effects. The main difference is that while propositional attributes are logical conjunctions, metric attributes also involve numerical expressions. This approach is similar to that taken by other AI planners that deal with resources. In particular, the knowledge representation scheme recently suggested by Koehler [Koe98] to deal with resources is strikingly similar to PR-STRIPS.

PR-STRIPS.

In PR-STRIPS, the world states are described in terms of attributes which may be propositional or metric.

Definition 1 (State): *A PR-STRIPS state is a 2-tuple* $< S_p, S_m >$ *where* S_p *denotes propositional attributes and* S_m *denotes metric attributes.*

Definition 2 (Propositional Attribute): *A propositional attribute is a 2-tuple* $< n, v >$ *where* n *is the symbol denoting the proposition name and* v *is the proposition value.*

Definition 3 (Metric Attribute): *A metric attribute is a formula* $< \beta, l >$ *where* β *is a symbol denoting a resource name and* l *is a real number denoting the amount or level of that resource.*

Definition 4 (Metric Effect): *The metric effect of an action* α *is a formula* $< \beta, F_{\alpha\beta} >$ *where* β *is a resource and* $F_{\alpha\beta}$ *is a metric effect function defined over all possible bindings of* α*'s parameters* $< p_1, \ldots, p_n >$.

---

[1] Williamson's original formulation of R-STRIPS also allowed for partially satisfiable goals. PR-STRIPS restricts its goal expressions to propositional formulas that have to be completely satisfied because PIP does not reason with metric and partially satisfiable goals. Williamson also defines the outcome of a plan to include the intermediate states as well as the final state.

Definition 5 (Action Schema): *A PR-STRIPS action schema is a five-tuple* $\alpha =< \alpha_n, \alpha_v, \alpha_{pp}, \alpha_{pe}, \alpha_{me} >$ *where*

- α_n *denotes the symbolic name,*
- α_v *is a list of variable parameters,*
- α_{pp} *denotes preconditions.*
- α_{pe} *denotes propositional effects, and*
- $\alpha_{me} = \{< \beta, F_{\alpha\beta} > \mid$ *for each resource* $\beta\}$ *is a set of metric effects.*

Definition 6 (Ground Action): *A ground action is an action-schema in which all variables have been bound to object symbols in the domain.*

A ground action represents a mapping between world states. This mapping is defined over those states in which the action is viable.

Definition 7 (Viability of an action): *An action* α *is viable in a state* $S =< S_p, S_m >$ *if its preconditions are present in that state i.e.,* $\alpha_{pp} \subseteq S_p$

Definition 8 (Action Execution): *The execution of a viable action* α *in a world state* $S =< S_p, S_m >$ *is a new world state* $S' =< S'_p, S'_m >$ *such that*

$$S'_p = \alpha_{pe} \cup S_p$$

and

$$S'_m = \{< \beta, l + F_{\alpha\beta} > \mid < \beta, l > \in S_m\}$$

i.e., the new state is obtained by adding the propositional effects of the action to the next state and the levels of resources are computed by taking out the amounts of resources consumed.

Definition 9 (Plan): *A plan is an ordered sequence of ground action schemas.*

Definition 10 (Plan Viability): *A plan* $p =< a_1, \ldots, a_n >$ *is viable in state* S_i *if each action* a_i, $1 \leq i \leq n$, *is viable in the state* S_i *where* $S_i = a_{i-1}(S_{i-1})$ *for all* $i > 0$ *and* $S_0 =$ *initial-state.*

Definition 11 (Plan Outcome): *The outcome of a plan is the final-state achieved by executing the plan in the initial-state.*

Definition 12 (Plan Quality): *The quality of a plan is the evaluation of the value function computed by substituting amounts of the metric resources consumed by the plan.*

Using PR-STRIPS, the domains discussed in Chapter 1 can be represented and reasoned with. Appendices A, B and C show the PR-STRIPS encodings of Transportation, Softbot and Process-planning domains discussed in the first chapter. These domains are also used in empirical evaluations of the PIP framework presented in the second part of Chapter 4.

3.2 Architecture and Algorithms

PIP has four main components as shown in Figure 3.1. The first is a causal-link partial-order planner (POP) similar to SNLP [MR91]. The task of the planning component is the generation of the default planning episode. The second component is the *model planning episode* generation component. It generates the better quality (i.e., better quality than the system's default plan) model planning episode. The idea is to compare these two planning episodes to discover rules that, if followed, would allow the system to generate the model planning episode. The differences between the two planning episodes are therefore learning opportunities for identifying planning decisions that lead to higher quality model plan(s). PIP maintains a *rule library* in which rules are indexed for easy retrieval.

Figure 3.2 shows PIP's high level algorithm. Each step of PIP's high level algorithm is illustrated next with the help of the following example.

Example: Consider the transportation problem shown in Figure 3.3. It involves transporting two objects; *o*1 and *o*2. The initial-state is described by both metric attributes (such as *time* and *money* that indicate the levels of these resources in the initial state) and propositional attributes (such as *at-object* and *at-truck* that indicate locations of these objects in the initial

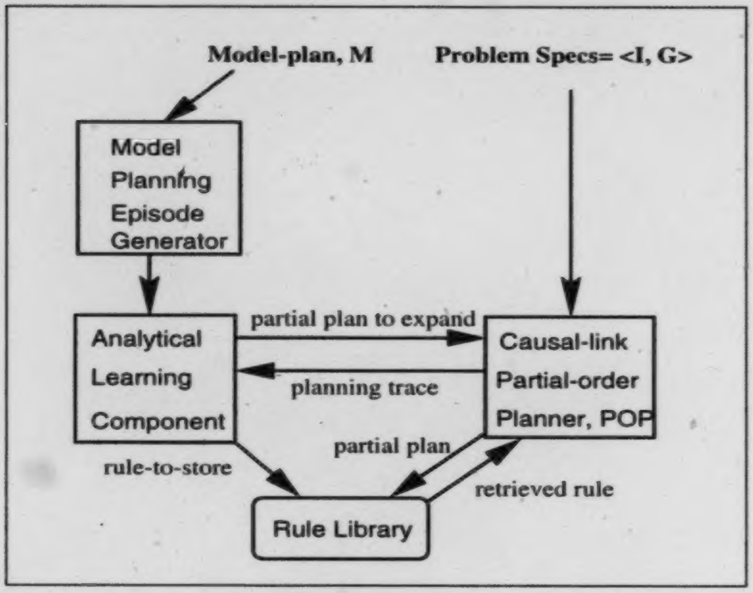

Model-plan, M
Problem Specs= <I, G>
Model Planning Episode Generator
Analytical Learning Component
partial plan to expand
planning trace
Causal-link Partial-order Planner, POP
partial plan
rule-to-store
retrieved rule
Rule Library

Figure 3.1: PIP's architecture. The box with round edges represents PIP's rule library while other components are represented by boxes with square edges. The arrows between the boxes represent flow of information and control between various components.

state). Figure 3.3 also shows two different plans for this problem. PIP's default planner produces the plan to use truck *tr*1 to transport both objects while the higher quality model plan uses the plane to fly object *o*1 from airport *ap*1 to airport *ap*2 and uses truck *tr*2 to transport object *o*2. This indicates that PIP's default planner does not possess the correct rationale for applying the good (the planning decisions that can lead to the model plan) or the bad planning decisions (the planning decisions that can lead to a lower quality plan)[2]. The objective of PIP's learning algorithm is to learn these rationales so that it can take good planning decisions and avoid bad planning decisions in similar situations in the future.

---

[2] Had PIP possessed the correct rationale for applying the bad planing decisions, it would not have applied them in the current situation.

```
Input:   - Problem description in terms of initial-state I and goals G
Output:  - A set of rules
--------------------------------------------------------------------
1- Use the existing planner to generate the system's plan P and system's
     planning trace Ptr for this problem.
2- If in apprenticeship mode then
       - ask the user for a model plan M
       - infer the model constraint-set Mc from M
   else
       - generate a better quality (than P) plan M and model
         constraint-set Mc
3- Identify learning opportunities by comparing the system's planning
     trace Ptr with the model constraint-set Mc
4- Learn a rule from each learning opportunity  and store it.
```

Figure 3.2: PIP's high level algorithm.

3.2.1 Step 1: Generating the Default Planning Episode

Given the domain knowledge (i.e., PR-STRIPS encoding of domain actions and a quality function) and a planning problem (i.e., initial state and goals encoded in PR-STRIPS), PIP's first step is generation of the *default planning episode* using PIP's default planner. The default planning episode is defined as the default plan and the default planning trace, i.e., a record of the planning decisions taken by PIP's default planner to produce the default plan. PIP's default planner refers to the planning component plus the existing rule library. The planner consults its rule library to see which rules, if any, are applicable in the current planning situation. It uses the rules, if any are retrieved, to produce the default plan and the default planning trace.

PIP's default planner POP is a variation of SNLP [MR91] with the following two differences:

- In POP, the variable binding and propagation constraints are implicitly handled and are not explicitly represented[3].

---

[3] This is possible because POP is implemented in Prolog which allows the variable instantiation, propagation and enforcement to be handled by the compiler.

Initial-state: {at-object(o1, ap1), at-object(o2, ap2), at-truck(tr1, ap1),
at-truck(tr2, ap2), at-plane(pl1, ap1), same-city(ap1, po1),
same-city(po1,ap1), same-city(ap2,po2),same-city(po2,ap2),
position(ap1, 10), position(po1, 15), position(ap2, 100),
position(po2, 110), money(1000), time(0)}

Goals: {at-object(o1, ap2), at-object(o2, po2)}

System's Default Plan	Model Plan
load-truck(o1, tr1, ap1)	load-plane(o1, pl1, ap1)
drive-truck-acities(tr1, ap1, ap2)	fly-plane(pl1, ap1, ap2)
unload-truck(o1, tr1, ap2)	unload-plane(o1, pl1, ap2)
load-truck(o2, tr2, ap2)	load-truck(o2, tr2, ap2)
drive-truck(tr2, ap2, po2)	drive-truck(tr2, ap2, po2)
unload-truck(o2, tr2, po2)	unload-truck(o2, tr2, po2)

Figure 3.3: Problem 1: A Transportation planning domain. The goal is to have the objects o1 transported from airport 1 (ap1) to the airport 2 (ap2) and the object o2 transported from the airport 2 (ap2) to the post-office 2 (po2).

- POP has an extra step (Step 2.2.1 in the POP algorithm shown in Figure 3.4) added to ensure that any previously learned search control rules matching a partial-plan being refined are retrieved and used to guide its refinement. POP is still complete because if no search-control rule is available to guide the planning process, POP reverts to the generative partial-order planning algorithm which is complete [MR91].

A partial plan P in POP is a five-tuple $< A_P, O_P, L_P, E_P, C_P >$ where

- A_P is the set of actions,
- O_P is the set of ordering constraints on the actions in A_P,
- L_P is the set of *causal links.* A causal link $p \xrightarrow{e} c$ between the producer action p and the consumer action c (i.e., producer and consumer of effect e) is said to exist as long as p comes before c and no action t can come between p and c (in all linearizations of the plan) that deletes e,
- E_P is set of effects of actions in A_P, and

- C_P is the set of open conditions. C_P keeps a list of the pending goals/subgoals.

A dummy action *end* is considered to have the problem goals as its preconditions and a dummy action *start* is considered to have the conditions specified in the initial-state as its effects. As shown in Figure 3.4, POP's first step is to add the actions *start* and *end* to the action-set A_P, their preconditions and effects to its open conditions-set C_P and effect-set E_P respectively, and the ordering constraint $start \succ end$ to the ordering constraint-set O_P to initialize the partial plan P.

A partial plan is considered to have *flaws* and planning is considered to be the process of refining it until all its flaws are eliminated. If the plan contains some open conditions that are not supported by any causal link it is said to contain an *open condition flaw.* It is said to contain an *unsafe link flaw* if it contains a causal link constraint, and an action (called the *threat*) that can possibly come between the producer and the consumer of the causal link and delete the condition being supported by the causal link. If the flaw is an unsafe link (Step 2.2.2.1 of POP algorithm, Figure 3.4), involving a causal link $s \xrightarrow{c} w$ and a threatening action t, POP resolves it by either promoting t to come after w or by demoting it to come before s. If the flaw is an open condition (Step 2.2.2.2 of POP algorithm, Figure 3.4), POP resolves it by either using an effect of an existing action (*establishment*) or by adding a new action (*add-action*). Thus there are four types of decision nodes in a POP search-tree: establishment, action-addition, promotion and demotion.

The default plan produced by POP for the transportation problem is shown in Figure 3.3. POP's planning trace for this problem (shown in Figures 3.7 and 3.8) shows a record of all the planning decisions POP took to refine this plan. The partial plan being refined are shown in the square boxes and the lines connecting the boxes represent the planning decisions that PIP took to transform a partial plan n into the partial plan $n + 1$ (shown below n). All the satisfied preconditions of an action A are shown by the arrows pointing towards A and originating from the actions supplying those preconditions. All the unsatisfied preconditions of each action are displayed next to the action

```
POP (Init-state, Goals, Action-schemas)
    1- [Initialize P]
        - $A_P \longleftarrow \{start, end\}$
        - $O_P \longleftarrow \{start \succ end\}$
        - $L_P \longleftarrow \{\}$
        - $E_P \longleftarrow$ Init-state
        - $C_P \longleftarrow$ Goals
        - Ptr $\longleftarrow$ {}
    2- return refine(P, Ptr)

refine (P, Ptr)
    2.1- If not flaw(P) then
        2.1.1- Done
    2.2-else
        2.2.1- if R $\longleftarrow$ retrieve-a-rule(P) then
            2.2.1.1- replay(P, Ptr, R)
        2.2.2- else
            2.2.2.1-if unsafe-links(P, Threats) then
                    if resolve-threats(Threats, P, Ptr) then
                        $(P, Ptr) \longleftarrow$ resolve-threats(Threats, P, Ptr)
                        return refine(P, Ptr)
                    else
                        fail
            2.2.2.2- if $\exists\, c^{a_i} \in C_P$ then
                    if resolve-an-open-cond($c^{a_i}$, P, Ptr) then
                        $(P, Ptr) \longleftarrow$ resolve-an-open-cond($c^{a_i}$, P, Ptr)
                        return refine(P, Ptr)
                    else
                        fail
```

Figure 3.4: The POP algorithm (Step 1 of Algorithm 1). Comments are enclosed in square brackets.

```
**resolve-an-open-cond** ($c^{a_i}$, P, Ptr)
    **If** $\exists$ an action $a_j \in A_P$ that adds c **then**
            [establish]
            - $L_P \longleftarrow L_P \cup \{a_j \xrightarrow{c} a_i\}$
            - $O_P \longleftarrow O_P \cup \{a_j \succ a_i\}$
            - Ptr $\longleftarrow$ Ptr $\cup\{a_j \xrightarrow{c} a_i\}$
            **return** (P, Ptr)
    **else**
        **if** find-a-new-action($a_j$) that adds c **then**
            [add-action $a_j$]
            - $A_P \longleftarrow A_P \cup \{a_j\}$
            - $L_P \longleftarrow L_P \cup \{a_j \xrightarrow{c} a_i\}$
            - $O_P \longleftarrow O_P \cup \{a_j \succ a_i\}$
            - Ptr $\longleftarrow$ Ptr $\cup\{a_j \xrightarrow{c} a_i\}$
            **return** (P, Ptr)
        **else**
            **return** failure

**resolve-threats** ($\{t1, t2, \ldots, tn\}$, P, Ptr)
    **If** resolve-a-threat(t1, P, Ptr) then
        (P, Ptr) $\longleftarrow$ resolve-a-threat(t1, P, Ptr)
        (P, Ptr) $\longleftarrow$ resolve-threats($\{t2, \ldots, tn\}$, P, Ptr)
    **else**
        **return** failure

**resolve-a-threat** ($(t, p \xrightarrow{c} c)$, P, Ptr)
    **If** consistent($t \succ p$) **then**
            [promote $t$]
            - $O_P \longleftarrow O_P \cup \{t \succ p\}$
            - Ptr $\longleftarrow$ Ptr $\cup\{t \succ p\}$
            - **return** (P, Ptr)
        **else**
            **If** consistent($c \succ t$) **then**
                    [demote $t$]
                    - $O_P \longleftarrow O_P \cup \{c \succ t\}$
                    - Ptr $\longleftarrow$ Ptr $\cup\{c \succ t\}$
                    - **return** (P, Ptr)
            **else**
                **return** failure
```

Figure 3.5: Continuation of the POP algorithm. c^{a_i} denotes precondition c of action a_i. Comments are enclosed in square brackets.

```
retrieve-a-rule ($P$)
    Best-quality $\longleftarrow$ 0
    R $\longleftarrow$ NULL
    For each rule $R_i \in$ rule-library do
        if open-conditions($R_i$) $\subseteq C_P$ and effects($R_i$) $\subseteq E_P$ and
        quality($R_i$) > Best-quality then
            Best-quality $\longleftarrow$ quality($R_i$)
            $R \longleftarrow R_i$
    return $R$.

replay (P, Ptr, $\{c_1, c_2, \ldots, c_n\}$)
    For all $c_i$ do
        $P \longleftarrow$ add-constraint($c_i$, $P$)
        Ptr $\longleftarrow$ Ptr $\cup\ c_i$
    return (P, Ptr).

unsafe-links (P, Threats)
    if $\exists\ not(e)^t \in E_P$ and $p \xrightarrow{e} c$ and
        $\nexists\ (t \succ p$ or $c \succ t)$ then
            $\forall\ not(e)^t \in E_P$ and $p \xrightarrow{e} c$ and
            $\nexists\ (t \succ p$ or $c \succ t)$ do
                Threats $\longleftarrow$ Threats $\cup \{(t, p \xrightarrow{e} c)\}$
                return true
    else
        return false

add-constraint(c, P)
    if $c = a_1 \succ a_2$ then [if $c$ is an ordering constraint]
        $O_P \longleftarrow O_P \cup \{a_1 \succ a_2\}$
    else if $c = a_1 \xrightarrow{e} a_2$ then
        $O_P \longleftarrow O_P \cup \{a_1 \succ a_2\}$
        $L_P \longleftarrow L_P \cup a_1 \xrightarrow{e} a_2$
    return P.
```

Figure 3.6: Continuation of the POP algorithm. The expressions open-conditions(R) and effects-needed(R) denote open-conditions field and effects field of the rule R respectively.

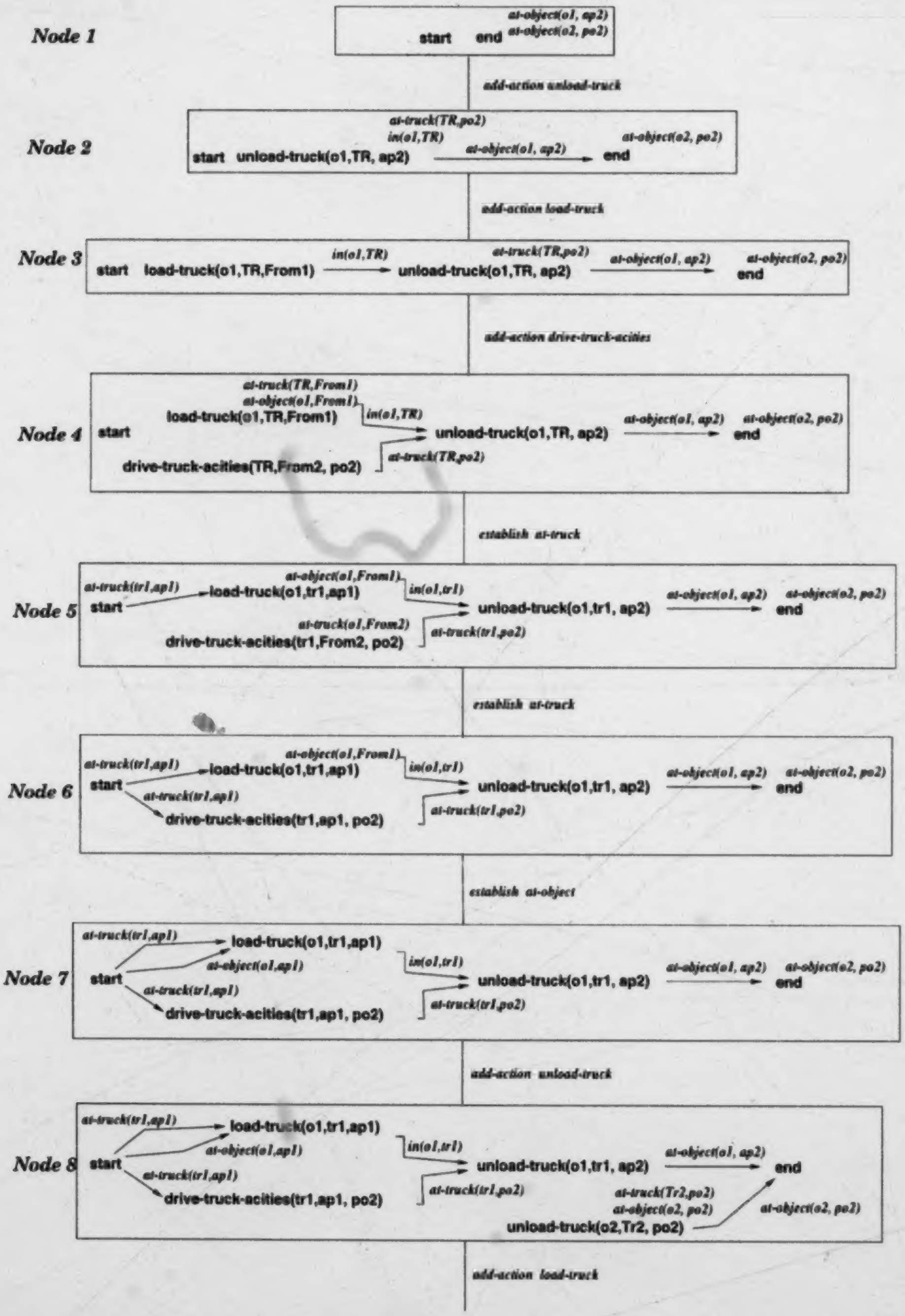

Node 1
start
end
at-object(o1, ap2)
at-object(o2, po2)
add-action unload-truck
Node 2
start
unload-truck(o1,TR, ap2)
at-truck(TR,po2)
in(o1,TR)
at-object(o1, ap2)
end
at-object(o2, po2)
add-action load-truck
Node 3
start
load-truck(o1,TR,From1)
in(o1,TR)
unload-truck(o1,TR, ap2)
at-truck(TR,po2)
at-object(o1, ap2)
end
at-object(o2, po2)
add-action drive-truck-acities
Node 4
start
at-truck(TR,From1)
at-object(o1,From1)
load-truck(o1,TR,From1)
in(o1,TR)
drive-truck-acities(TR,From2, po2)
at-truck(TR,po2)
unload-truck(o1,TR, ap2)
at-object(o1, ap2)
end
at-object(o2, po2)
establish at-truck
Node 5
start
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
at-object(o1,From1)
in(o1,tr1)
at-truck(o1,From2)
drive-truck-acities(tr1,From2, po2)
at-truck(tr1,po2)
unload-truck(o1,tr1, ap2)
at-object(o1, ap2)
end
at-object(o2, po2)
establish at-truck
Node 6
start
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
at-object(o1,From1)
in(o1,tr1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
at-truck(tr1,po2)
unload-truck(o1,tr1, ap2)
at-object(o1, ap2)
end
at-object(o2, po2)
establish at-object
Node 7
start
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
at-object(o1,ap1)
in(o1,tr1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
at-truck(tr1,po2)
unload-truck(o1,tr1, ap2)
at-object(o1, ap2)
end
at-object(o2, po2)
add-action unload-truck
Node 8
start
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
at-object(o1,ap1)
in(o1,tr1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
at-truck(tr1,po2)
unload-truck(o1,tr1, ap2)
at-object(o1, ap2)
end
at-truck(Tr2,po2)
at-object(o2, po2)
unload-truck(o2,Tr2, po2)
at-object(o2, po2)
add-action load-truck

Figure 3.7: Default planning trace for the transportation example problem (Problem 1 shown in Figure 3.3).

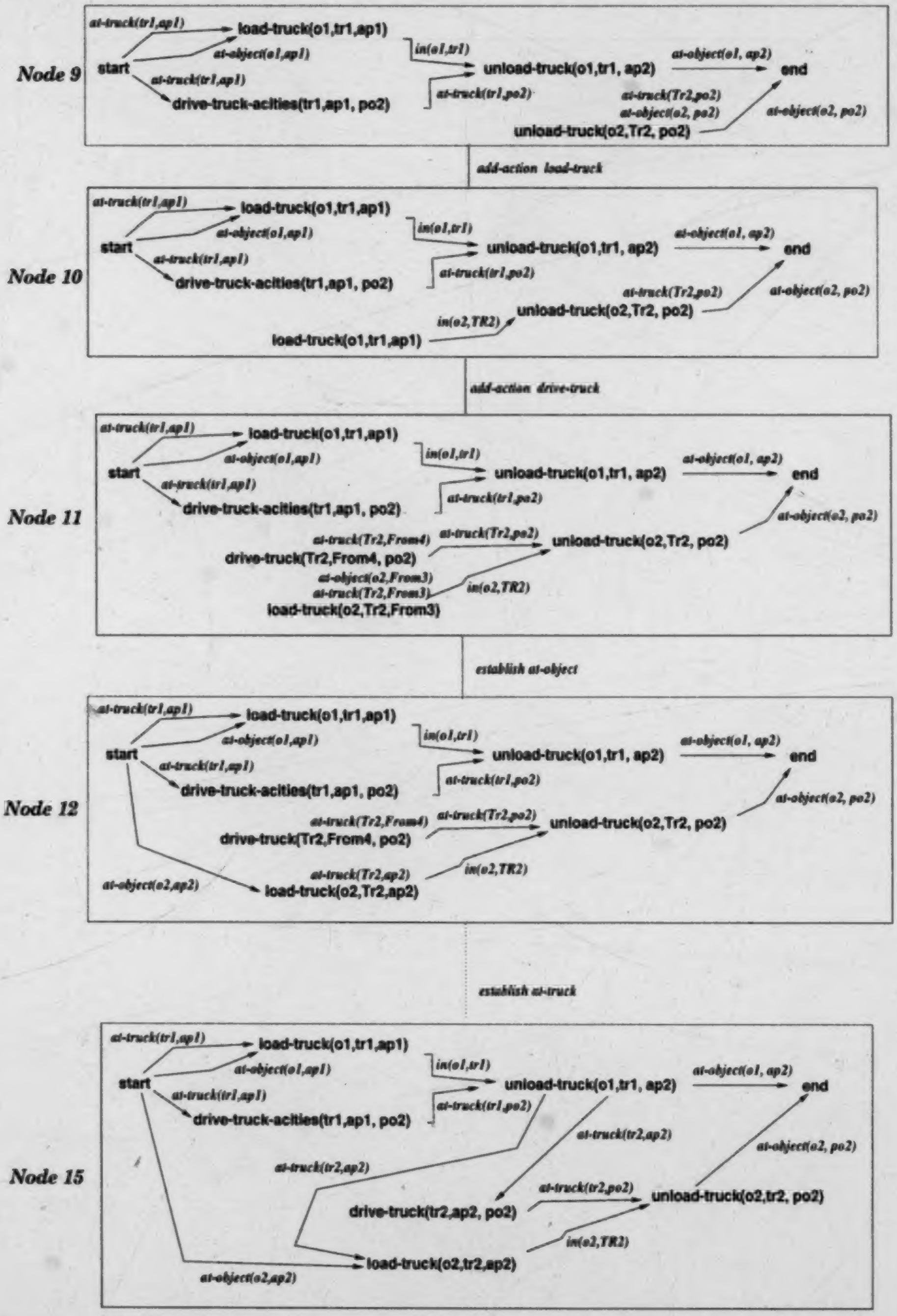

Node 9
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
start
at-object(o1,ap1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
in(o1,tr1)
unload-truck(o1,tr1, ap2)
at-truck(tr1,po2)
at-object(o1, ap2)
end
at-truck(Tr2,po2)
at-object(o2, po2)
unload-truck(o2,Tr2, po2)
at-object(o2, po2)
add-action load-truck
Node 10
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
start
at-object(o1,ap1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
in(o1,tr1)
unload-truck(o1,tr1, ap2)
at-truck(tr1,po2)
at-object(o1, ap2)
end
at-truck(Tr2,po2)
unload-truck(o2,Tr2, po2)
at-object(o2, po2)
in(o2,TR2)
load-truck(o1,tr1,ap1)
add-action drive-truck
Node 11
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
start
at-object(o1,ap1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
in(o1,tr1)
unload-truck(o1,tr1, ap2)
at-truck(tr1,po2)
at-object(o1, ap2)
end
at-object(o2, po2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
drive-truck(Tr2,From4, po2)
unload-truck(o2,Tr2, po2)
at-object(o2,From3)
at-truck(Tr2,From3)
in(o2,TR2)
load-truck(o2,Tr2,From3)
establish at-object
Node 12
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
start
at-object(o1,ap1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
in(o1,tr1)
unload-truck(o1,tr1, ap2)
at-truck(tr1,po2)
at-object(o1, ap2)
end
at-object(o2, po2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
drive-truck(Tr2,From4, po2)
unload-truck(o2,Tr2, po2)
at-truck(Tr2,ap2)
in(o2,TR2)
at-object(o2,ap2)
load-truck(o2,Tr2,ap2)
establish at-truck
Node 15
at-truck(tr1,ap1)
load-truck(o1,tr1,ap1)
start
at-object(o1,ap1)
at-truck(tr1,ap1)
drive-truck-acities(tr1,ap1, po2)
in(o1,tr1)
unload-truck(o1,tr1, ap2)
at-truck(tr1,po2)
at-object(o1, ap2)
end
at-truck(tr2,ap2)
at-object(o2, po2)
at-truck(tr2,ap2)
at-truck(tr2,po2)
drive-truck(tr2,ap2, po2)
unload-truck(o2,tr2, po2)
in(o2,TR2)
load-truck(o2,tr2,ap2)
at-object(o2,ap2)

Figure 3.8: Continuation of Figure 3.7: default planning trace for Example 3.1. The dotted line between Nodes 12 and 15 indicates that Nodes 13-14 have been omitted for brevity.

(on its right top end). The root node (Node 1) shows the initial-plan and the leaf node (Node 15) shows the completely refined partial plan (i.e., a complete plan).

The key idea behind PIP's algorithm is that partial-order planning is a refinement process i.e., the process of adding constraints to a partial plan. Each planning decision to resolve a threat can be seen as adding an ordering constraint and each *add-action/establish* decision can be seen as adding both an ordering constraint as well as a causal link constraint. For instance, the planning decision recorded at the first node in Figure 3.7 adds the ordering constraint $start \succ end$. Node 2 adds the causal link constraint $unload\text{-}truck(o1, TR, ap2) \xrightarrow{at\text{-}objects(o1,ap2)} end$ and the ordering constraint $unload\text{-}truck(o1, TR, ap2) \succ end$. Similarly, each planning decision can be seen as adding some constraints to the partial plan. This means that all the information contained in the planning trace shown in Figures 3.7 and 3.8 can be represented as an ordered constraint-set as shown in Figure 3.9. This observation allows us to define a planning trace as an *ordered* set of constraints (causal-link and the ordering constraints stored in the order in which they were added by the planner).

After generating the default planing episode, PIP calls its model planning episode generator to generate a model planning episode. This model planning episode is then compared with the default planning episode.

3.2.2 Step 2: Generating the Model Planning Episode

The model planning episode is defined as a better quality[4] model plan and the *model planning constraint-set* (or simply the *model constraint-set*). The model constraint-set is an unordered set of causal-links and ordering constraints that are compatible with the model plan. PIP only needs an unordered set (as opposed to an ordered constraint-set i.e., a planning trace) because it does not learn in what *order* the planning decisions should be taken (i.e., in what

---

[4] Current version of PIP only learns when the alternative plan is of better quality than the default plan. PIP can be easily modified to learn from lower quality alternative plans. Off course, instead of learning how to plan, in this case PIP will learn how *not to plan*.

1. unload-truck(o1, TR, ap2) $\xrightarrow{\text{at-object(o1, ap2)}}$ end
2. load-truck(o1, TR, From1) $\xrightarrow{\text{In(o1, TR)}}$ unload-truck(o1, TR, ap2)
3. drive-truck-acities(TR, From1, ap2) $\xrightarrow{\text{at-truck(TR,ap2)}}$ unload-truck(o1, TR, ap2)
4. start $\xrightarrow{\text{at-truck(tr1, ap1)}}$ load-truck(o1, tr1, ap1)
5. start $\xrightarrow{\text{at-truck(tr1, ap1)}}$ drive-truck-acities(tr1, ap1, ap2)
6. start $\xrightarrow{\text{at-object(o1, ap1)}}$ load-truck(o1, tr1, ap1)
7. unload-truck(o2, tr2, po2) $\xrightarrow{\text{at-object(o2, po2)}}$ end
8. load-truck(o2, tr2, ap2) $\xrightarrow{\text{In(o1, tr2)}}$ unload-truck(o2, tr2, po2)
9. drive-truck(tr2, ap2, po2) $\xrightarrow{\text{at-truck(tr2, po2)}}$ unload-truck(o2, tr2, po2)
10. start $\xrightarrow{\text{at-object(o2, ap2)}}$ load-truck(o2, tr2, ap2)
11. start $\xrightarrow{\text{at-truck(o2, ap2)}}$ load-truck(o2, tr2, ap2)

Figure 3.9: Ordered constraint-set corresponding to the planning trace shown in Figures 3.7 and 3.8. Only causal-link constraints are shown here.

order to resolve the pending flaws[5]) but only *what* different planning decisions should be taken to generate a higher quality plan.

The model planning episode generator consists of an alternative planner and an infer-constraints module. If PIP is in the autonomous learning mode, then the alternative planner is used to produce the model plan and the model constraint-set. If the system is in the apprenticeship mode, then the user is asked for a model plan but not the model constraint-set[6]. PIP has to infer these constraints from the model plan. Essentially, PIP must infer some aspects of the planning decisions that would be consistent with a particular plan, because it compares planning decision traces and not totally ordered plans. A naive way of inferring those constraints would be to search exhaustively until the model plan is produced. However, this is extremely inefficient and in the worst case requires searching the entire search space. A more efficient method is to use the model plan as a guide to limit the search. Figure 3.10 shows the *infer-constraints* algorithm used by PIP. This algorithm differs from the basic partial order planning algorithm in its implementation of the *add-action* procedure. PIP uses this algorithm to compute the set of model constraints that would have been imposed by PIP's default planner had it produced the model plan. Figure 3.11 shows the constraints inferred by PIP from the model plan shown in Figure 3.3.

---

[5] This is not to say that the flaw selection order does not impact a partial-order planner's performance. Indeed, there has been some important work done by [Wil96] to study the impact of flaw selection strategies on the performance of partial-order planners.

[6] Most apprenticeship systems [MMST93] assume that the user only provides the final solution. The reason is that if these systems are to be deployed in the real world planning situations, then we cannot assume that their users know how the problem solving algorithm works.

The problem is that a model plan may be compatible with more than one constraint-set. This happens when more than one effect is available to satisfy an open-condition. Since the learning complexity of PIP's learning algorithm depends on the number of conflicting choice points, PIP can learn more efficiently from a constraint-set that leads to smaller number of conflicting choice points than a constraint-set that leads to a larger number of conflicting choice points[7]. Hence the optimal learning strategy for PIP would be to compute the constraint-set that leads to the smallest number of conflicting choice points. A naive algorithm to do that would be to compute all the constraint sets for a given model plan, find out the number of conflicting choice points generated by each constraint-set, and select the one that leads to the smallest number. If n is the plan length (i.e., the number of actions) then in the worst case, there may be n ways of resolving each goal (as each goal may be supplied by all the n actions). Each precondition of these actions may in turn be supplied by all the remaining actions. If m is the average number of preconditions that an action has in this domain then in the worst case the number of constraint sets compatible with the model plan is equal to $m^n \times n!$. Inferring all the constraint sets and computing the number of conflicting choice points generated by each constraint-set may be too costly. PIP uses a heuristic technique (shown in

---

[7] Prefering constraint sets that lead to fewer conflicting choice points (and hence fewer rules) is also a good heuristics for keeping PIP's rule library size small. A small rule library allows faster rule retrieval and hence is preferable.

```
Infer-constraints ($M$)
        [Initialize P]
        - $A_P \longleftarrow \{start, end\}$
        - $O_P \longleftarrow \{start \succ end\}$
        - $L_P \longleftarrow \{\}$
        - $E_P \longleftarrow$ Init-state
        - $C_P \longleftarrow$ Goals
    - return refine(P, M)


refine (P, M)
    1- If not flaw(P) then
        1.1- return success
    else
        if unsafe-links(P) then
            if resolve-threats(P) then
                (P, Ptr) $\longleftarrow$ refine(P, Ptr)
                    [call POP's refine shown in Figure 3.4]
            else
                fail
        if $\exists\ c^{a_i} \in C_P$ then
            if resolve-an-open-cond($c^{a_i}$, M) then
                return refine(P, M)
            else
                fail


resolve-an-open-cond ($c^{a_i}$, M)
    -If $\exists$ an action $a_j \in A_P$ that adds c then
            [establish]
            - $L_P \longleftarrow L_P \cup \{a_j \xrightarrow{c} a_i\}$
            - $O_P \longleftarrow O_P \cup \{a_j \succ a_i\}$
    else
        if $\exists$ an action $a_j \in M$ that has an effect $c$ then
            [add-action $a_j$]
            - $A_P \longleftarrow A_P \cup \{a_j\}$
            - $L_P \longleftarrow L_P \cup \{a_j \xrightarrow{c} a_i\}$
            - $O_P \longleftarrow O_P \cup \{a_j, \succ a_i\}$
        else
            fail
```

Figure 3.10: PIP's Infer-constraints algorithm. Comments are enclosed in square brackets.

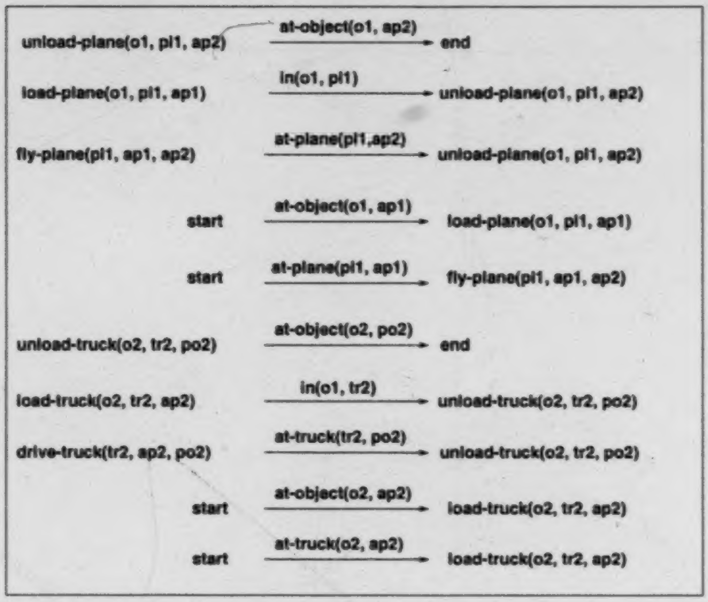

unload-plane(o1, pl1, ap2)
at-object(o1, ap2)
end
load-plane(o1, pl1, ap1)
in(o1, pl1)
unload-plane(o1, pl1, ap2)
fly-plane(pl1, ap1, ap2)
at-plane(pl1,ap2)
unload-plane(o1, pl1, ap2)
start
at-object(o1, ap1)
load-plane(o1, pl1, ap1)
start
at-plane(pl1, ap1)
fly-plane(pl1, ap1, ap2)
unload-truck(o2, tr2, po2)
at-object(o2, po2)
end
load-truck(o2, tr2, ap2)
in(o1, tr2)
unload-truck(o2, tr2, po2)
drive-truck(tr2, ap2, po2)
at-truck(tr2, po2)
unload-truck(o2, tr2, po2)
start
at-object(o2, ap2)
load-truck(o2, tr2, ap2)
start
at-truck(o2, ap2)
load-truck(o2, tr2, ap2)

Figure 3.11: Model constraint-set for Problem 1, i.e., the constraints inferred by PIP from the model plan for Problem 1.

Figure 3.10) for efficiently computing a model constraint-set. The heuristic is to keep the infer-constraints algorithm as close to PIP's default planning algorithm as possible. In general, this strategy leads PIP to constraint sets that lead to few conflicting choice points.

3.2.3 Step 3: Analytically Comparing the two Episodes

Given the system's default planning episode and the model planning episode, PIP needs to identify (a) the planning decisions that the default planner has taken differently to produce the model plan and (b) the conditions under which these planning decisions lead to a higher quality plan. The approach taken here is that the default planner lacked the knowledge about when to take these planning decisions and hence it took the bad planning decisions (i.e., the decisions that lead to the lower quality plan) when it should have applied the good planning decisions (the decisions that lead to a higher quality plan). The solution is not just to remember the good planning decision but to learn the *rationale* for both the good planning decision and the bad planning decision so that it can apply both in appropriate situations in the future[8]. In order to do this PIP needs to:

- identify the planning decision points where the default planner made a different choice than the model planner, and
- learn the rationale for applying the planning decisions so that the default planner can apply the correct planning decisions in future.

PIP's analytic learning component uses the ISL algorithm shown in Figure 3.12. The input to ISL is both the default planning trace (computed in Step 1 of Algorithm 1) and the model constraint-set (computed in Step 2 of Algorithm 1). Given this information, ISL looks for differences between two planning episodes that lead to plans of different quality. This is done by retracing the default planning-trace, looking for a planning decision that added a constraint that is *absent* from the model constraint-set. These decision points are labeled *conflicting choice points*. Each conflicting choice point indicates a possible opportunity to learn the rationale for applying a planning decision that potentially contributes to the production of a better quality plan in a class of problems similar to the current problem.

There are four types of conflicting choice points:

- *add-action—add-action* conflicting choice points. These conflicting choice points arise when the two planning episodes add different actions to resolve the same open-condition flaw.
- *add-action—establish* and *establish—add-action* conflicting choice points. These points arise when one planner adds a new action to resolve an open-condition flaw while the other planner sees that the open condition can be satisfied (*established*) using an existing action.
- *establish—establish* conflicting choice points. These conflicting choice points arise when both planners resolve an open-condition flaw using two *different* effects of existing action(s) to establish an open-condition.

---

[8] The reason for this is that the planning decisions that are good in the current context may turn out to be bad in the context of another partial plan.

```
**ISL** (*Default-trace, Model-constraint-set*)
    1- [Initialize P]
        - $A_P \longleftarrow \{start, end\}$
        - $O_P \longleftarrow \{start \succ end\}$
        - $L_P \longleftarrow \{\}$
        - $E_P \longleftarrow$ Init-state
        - $C_P \longleftarrow$ Goals
    2- find-conflicting-choice-points(P, *Default-trace, Model-constraint-set*).

**find-conflicting-choice-points** (P, $Dtr = \{d_1, d_2, \ldots d_n\}, Mc$)
    2.1 **if** $d_1 \in Mc$ **then**
        2.1.1 **if** $d_1 = p \xrightarrow{e} c$ **then**
                $L_P \longleftarrow L_P \cup \{p \xrightarrow{e} c\}$
                $O_P \longleftarrow O_P \cup \{p \succ c\}$
        2.1.2 **else**
                $O_P \longleftarrow O_P \cup \{d_1\}$
        2.1.3 find-conflicting-choice-points(P, $Dtr - \{d_1\}$, Mc)
    2.2 **else**
        2.2.1 mark this node as a conflicting choice point
        2.2.2 flaw $\longleftarrow$ flaw-resolved-by $(d_1, P)$
        2.2.3 Model-Consts $\leftarrow$ find-constraint-in-Mc-that-resolves(flaw,Mc)
        2.2.4 $L_P \longleftarrow L_P \cup$ causal-link(Model-Consts)
        2.2.5 $O_P \longleftarrow O_P \cup$ ordering-constraints(Model-Consts)
        2.2.6 (P, Dtr) $\longleftarrow$ refine(P, Dtr)
                [call POP's refine defined in Figure 3.4]
        2.2.7 find-conflicting-choice-points(P, $Dtr - \{d_1\}$, Mc)
```

Figure 3.12: The Intra-Solution Learning (ISL) Algorithm (Step 3 of Algorithm 1 of Figure 3.2). Comments are enclosed in square brackets.

```
**flaw-resolved-by** ($d$)
    **if** $d = a_1 \succ a_2$ **and** $p \xrightarrow{e} c \in$ unsafe-links(P) **and**
    ($a_i = p$ xor $a_i = c$, where i=1,2) **then**
        **return** $(a_i, p \xrightarrow{e} c)$
    **else if** $d = p \xrightarrow{e} c$ **then**
        **return** $(p \xrightarrow{e} c)$

**causal-link** (Mc)
    model-consts $\longleftarrow$ {}
    $\forall d_i \in$ Mc **do**
        **if** $d_i = p \xrightarrow{e} c$ **then**
            model-consts $\longleftarrow$ model-consts $\cup$ $d_i$
    **return** model-consts

**ordering-constraints** (Mc)
    model-consts $\longleftarrow$ {}
    $\forall d_i \in$ Mc **do**
        **if** $d_i = a_1 \succ a_2$ **then**
            model-consts $\longleftarrow$ model-consts $\cup$ $d_i$
    **return** model-consts

**find-constraint-in-Mc-that-resolves** (flaw, Model-consts)
    **if** flaw $= p \xrightarrow{e} c$ **then**
        find $p' \xrightarrow{e} c \in$ Model-consts
        **return** $p' \xrightarrow{e} c \in$
    **else if** flaw $= (t, p \xrightarrow{e} c)$ **then**
        **if** $t \succ p \in$ Model-consts **then**
            **return** $t \succ p$
        **else**
            **return** $c \succ t$
```

Figure 3.13: Continuation of the Intra-Solution Learning (ISL) Algorithm.

- *promote–demote* and *demote–promote* conflicting choice points.[9] These points arise when one planner resolves a threat by promoting the threatening action to come before the producer of the threatened causal link while the other planner resolves the same flaw by demoting the threatening action to come after the consumer of the threatened causal-link.

Example: Given the default planning trace shown in Figures 3.7 and 3.8 and the model constraint-set shown in Figure 3.11, ISL retraces the default planning trace (shown in Figure 3.14) looking for a planning decision that adds a constraint not present in the model constraint-set. Node 1 in Figure 3.14 is one such node where the default planner resolves the open-condition flaw *at-object(o1,ap2)*$_{end}$ by performing *add-action: unload-truck(o1,TR,ap2)*, which adds the causal-link $unload\text{-}truck(o1, TR, ap2) \xrightarrow{at\text{-}obj(o1,ap2)} end$ to the partial plan. But this causal-link is not in the model constraint-set for this problem shown in Figure 3.11. The model constraint-set contains a causal link $unload\text{-}plane(o1, pl1, ap2) \xrightarrow{at\text{-}object(o1,ap2)} end$. In other words, the model planner resolved the precondition *at-object(o1,ap2)*$_{end}$ by *add-action: unload-plane(o1, pl1, ap2)*. Hence, Node 1 is labeled as an *add-action–add-action* type conflicting choice point.

Learning a single search control rule that ensures the application of the model planning decision at this point may turn a low-quality plan into a higher-quality plan, but it is rather unlikely that this was the only reason for the difference in quality between the default plan and the model plan. There may be more opportunities to learn what other decisions lead to a better quality plan for the same problem. To identify the other planning decisions whose rationale the default planer lacks, ISL adds the constraint added by the model plan at this point to the partial plan being refined (Steps 2.2.4 and 2.2.5 of ISL). Once the higher-quality plan's planning decision has been applied to the partial plan being refined, ISL calls the default planner *again to re-plan from that point on* (Step 2.2.6 of ISL). A new default plan and a new default trace

---

[9] The routine for learning from *promote–demote* or *demote–promote* conflicting choice points was never implemented. This would not have affected PIP's performance on any of the domains reported in the next two chapters because such opportunities never arise in any of those domains. It appears that such learning opportunities are very rare.

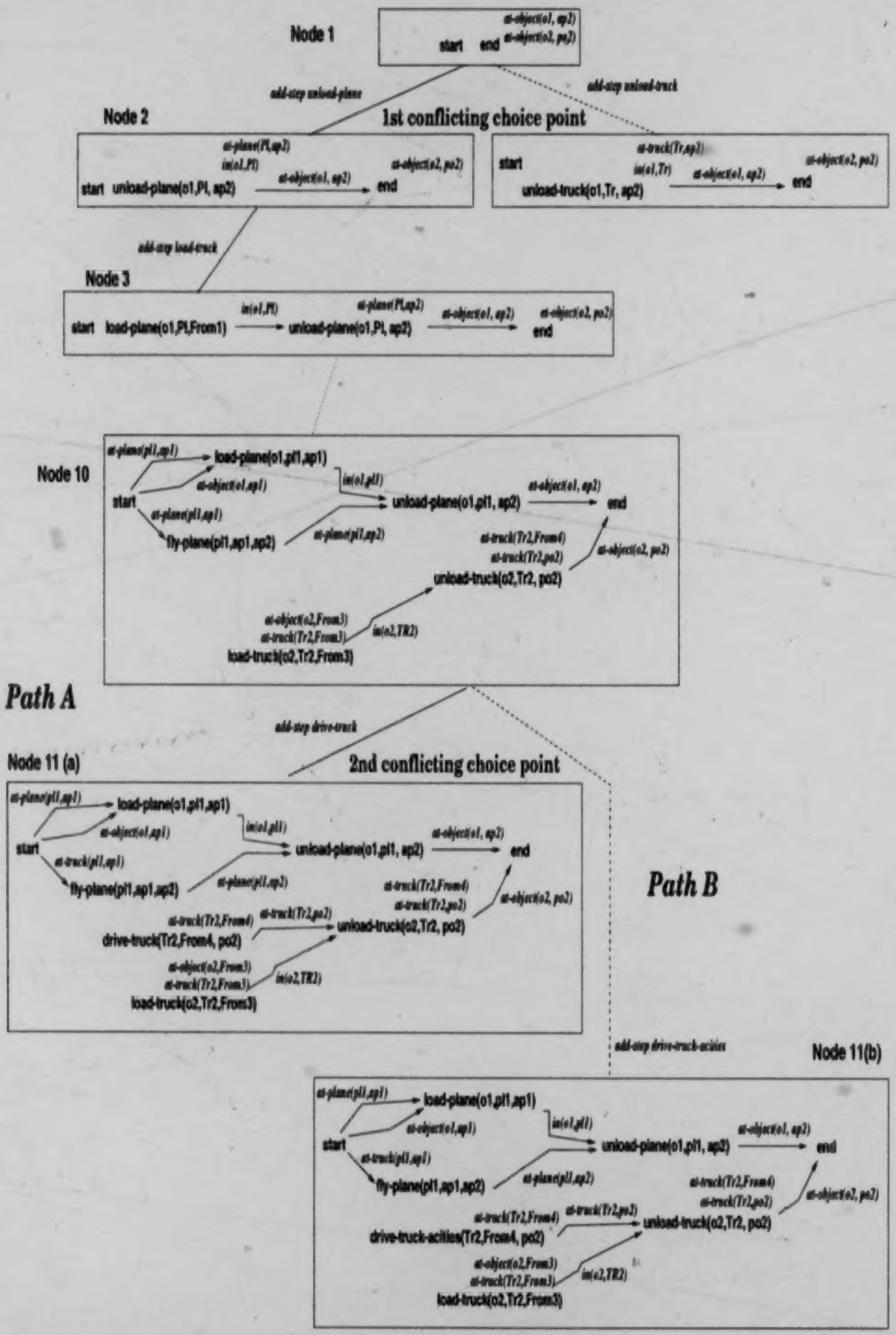

Node 1
start
end
at-object(o1, ap2)
at-object(o2, po2)
add-step unload-plane
add-step unload-truck
1st conflicting choice point
Node 2
start
unload-plane(o1,Pl, ap2)
at-plane(Pl,ap2)
in(o1,Pl)
at-object(o1, ap2)
end
at-object(o2, po2)
start
unload-truck(o1,Tr, ap2)
at-truck(Tr,ap2)
in(o1,Tr)
at-object(o1, ap2)
end
at-object(o2, po2)
add-step load-truck
Node 3
start
load-plane(o1,Pl,From1)
in(o1,Pl)
unload-plane(o1,Pl, ap2)
at-plane(Pl,ap2)
at-object(o1, ap2)
end
at-object(o2, po2)
Node 10
at-plane(pl1,ap1)
load-plane(o1,pl1,ap1)
at-object(o1,ap1)
start
in(o1,pl1)
unload-plane(o1,pl1, ap2)
at-object(o1, ap2)
end
at-plane(pl1,ap1)
fly-plane(pl1,ap1,ap2)
at-plane(pl1,ap2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
at-object(o2, po2)
unload-truck(o2,Tr2, po2)
at-object(o2,From3)
at-truck(Tr2,From3)
in(o2,TR2)
load-truck(o2,Tr2,From3)
Path A
add-step drive-truck
Node 11 (a)
2nd conflicting choice point
at-plane(pl1,ap1)
load-plane(o1,pl1,ap1)
at-object(o1,ap1)
start
in(o1,pl1)
unload-plane(o1,pl1, ap2)
at-object(o1, ap2)
end
at-truck(pl1,ap1)
fly-plane(pl1,ap1,ap2)
at-plane(pl1,ap2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
at-object(o2, po2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
drive-truck(Tr2,From4, po2)
unload-truck(o2,Tr2, po2)
at-object(o2,From3)
at-truck(Tr2,From3)
in(o2,TR2)
load-truck(o2,Tr2,From3)
Path B
add-step drive-truck-acities
Node 11(b)
at-plane(pl1,ap1)
load-plane(o1,pl1,ap1)
at-object(o1,ap1)
start
in(o1,pl1)
unload-plane(o1,pl1, ap2)
at-object(o1, ap2)
end
at-truck(pl1,ap1)
fly-plane(pl1,ap1,ap2)
at-plane(pl1,ap2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
at-object(o2, po2)
at-truck(Tr2,From4)
at-truck(Tr2,po2)
drive-truck-acities(Tr2,From4, po2)
unload-truck(o2,Tr2, po2)
at-object(o2,From3)
at-truck(Tr2,From3)
in(o2,TR2)
load-truck(o2,Tr2,From3)

Figure 3.14: Conflicting choice point that leads to Path A (left), from the higher-quality plan, and to Path B (right), the lower-quality plan.

(that is the same as the initial trace up to the now-replaced conflicting choice point, and possibly different thereafter) is returned for this same problem, and the process of analyzing this new trace against the constraints of the higher-quality model plan is done again. This analysis may lead to more conflicting choice points (as indeed is the case with the example scenario shown in Figure 3.14: at Node 10 the new default planning episode makes a different choice than the model plan). Eventually, the default planner will generate a planning trace that is consistent with the constraint-set inferred for the higher-quality model plan. That ends the learning about plan quality that can be accomplished from that single training problem.

For any conflicting choice point, there are two different planning decision sequences that can be applied to a partial plan: the one added by the default planner (the *worse* planning decisions), and the other added by the model planner (the *better* planning decisions). The application of one set of planning decisions leads to a higher quality plan and the other to a lower quality plan. It would be possible to construct a rule that indicates that the planning decision associated with the better-quality plan should be taken if that same flaw is ever encountered again. However, this would ensure a higher-quality plan *only if* that decision's impact on quality was not contingent on other planning decisions that are "downstream" in the refinement process, i.e., further along the search path. Thus, some effort must be expended to identify the dependencies between a particular planning decision and other planning decisions that follow it.

To identify what downstream planning decisions are relevant to the decision at a given conflicting choice point, the following method is used. The open-conditions at the conflicting choice point and the two different planning decisions (i.e., the ones associated with the high quality model plan and the lower quality default plan) are labeled as *relevant.* The rest of the better-plan's trace and the rest of the worse-plan's trace are then examined, with the goal of labeling a subsequent planning decision q relevant if

- there exists a causal-link $q \xrightarrow{c} p$ such that p is a relevant action, or

- *q* binds an uninstantiated variable of a relevant open-condition.

For instance, consider again the first conflicting choice point at Node 1 shown in Figure 3.14. There are two open-condition flaws in the partial plan, but the flaw selected to be removed at this point is the open-condition *at-object(o1, ap2)*. Clearly, the decision ***add-action:** unload-plane(o1,Pl,ap2)* on Path A (left path) is relevant. Similarly, the decisions to ***add-action:** load-plane(o1,pl1,ap1)* and ***add-action:** fly-plane(pl1,ap1,ap2)* are relevant because they supply preconditions to the relevant action *unload-plane(o1,Pl,ap2)*. Further along Path A, the decision ***establish:** at-object(o1, ap1)* is relevant because it supplies a precondition to the relevant action *fly-plane(pl1,ap1,ap2)*. However, the planning decisions ***add-action:** unload-truck(o2, Tr2, po2)*, and ***add-action:** drive-truck(Tr2, From4, po2)* are not relevant because the open conditions they resolve are not relevant. The labeling process stops on reaching the leaf nodes and the two relevant planning decision sequences (for each conflicting choice point) are returned. ISL returns the two planning decision sequences shown in Figure 3.15 for the first conflicting choice point.

3.2.4 Step 4: Forming and Storing Domain Specific Rules

Once ISL identifies the relevant planning decisions associated with the way in which given flaw(s) were resolved differently for the higher-quality plan and the lower quality plan, a search control rule can be created. The first step is to generalize the planning decision sequences. This is done by (a) replacing all the planning actions not added by the planning decision sequence (such as the *start* and *end* in the planning decision sequences shown in Figure 3.14) with variables and (b) replacing all the constants (such as *ap*1, *ap*2, *o*1, and *pl*1) with variables. For instance, generalizing the planning decision sequences of Figure 3.15 leads to the planning decision sequences shown in Figure 3.16.

The two generalized decision sequences (corresponding to each conflicting choice point) returned by ISL are stored as two search control rules. If the conflicting choice point is at a decision point to resolve an open condition flaw, then for each decision sequence PIP stores:

Lower quality sequence

add-action: *unload-truck(o1,Tr,ap2)* to resolve *at-object(O, Y)*$_{end}$

add-action: *load-truck(o1,Tr,From2)* to resolve *in(o1,Tr)*$_{unload\text{-}truck}$

add-action: *drive-truck-acities(Tr,From2,ap2)* to resolve *at-truck(Tr,ap2)*$_{unload\text{-}truck}$

establish: *at-object(o1, From2)*$_{load\text{-}truck}$ with *at-object(O, X)*^{start}

establish: *at-truck(Tr,From2)*$_{drive\text{-}truck\text{-}acities}$ with *at-truck(Tr, X)*^{start}

establish: *neq(ap1,ap2)*$_{drive\text{-}truck\text{-}acities}$ with *neq(X, Y)*^{start}

Higher quality sequence:

add-action: *unload-plane(o1,Pl,ap2)* to resolve *at-object(o1,ap2)*$_{end}$

add-action: *load-plane(o1,Pl,From1)* to resolve *in(o1,Pl)*$_{unload\text{-}plane}$

add-action: *fly-plane(Pl,From1,ap2)* to resolve *at-plane(Pl,From1))*$_{unload\text{-}plane}$

establish: *at-object(o1, From)*$_{load\text{-}plane}$ with *at-object(O, X)*^{start}

establish: *at-plane(Pl,X))*$_{fly\text{-}plane}$ with *at-plane(Pl, X)*^{start}

establish: *neq(ap1,ap2)*$_{fly\text{-}plane}$ with *neq(X, Y)*^{start}

Figure 3.15: Two planning decision sequences identified by ISL for the first conflicting choice point shown in Figure 3.14. The notation Pre_{Act} indicates that Pre is a precondition of Action Act and the notation Eff^{Act} indicates that Eff is an effect supplied by the action Act.

- the open-condition flaws present in its partial plan that the relevant decision sequence removes. These become the open-condition field of the rule.
- the effects present in its partial plan that are required by the relevant decision sequence. These become the effect field of the rule.
- the quality value of the new subplan produced by the relevant decision sequence. This becomes the quality field of the rule.

This information is then stored in PIP's rule library and specifies the rationale for applying the planning decision sequence stored in the rule.

By examining the better quality planning decision sequence returned by ISL for the example transportation problem (shown in Figure 3.15), PIP's rule

Lower quality sequence
    add-action: *unload-truck(O,Tr,Y)* to resolve *at-object(O, Y)*$_{Act1}$
    add-action: *load-truck(O,Tr,X)* to resolve *in(O,L)*$_{unload-truck}$
    add-action: *drive-truck-acities(Tr,X,Y)* to resolve
        at-truck(Tr,Y))$_{unload-\ truck}$
    establish: *at-object(O, X)*$_{load-truck}$) with *at-object(O, X)*^{Act2}
    establish: *at-truck(Tr,X)*$_{drive-truck-acities}$) with *at-truck(Tr, X)*^{Act3}
    establish: *neq(X,Y)*$_{drive-truck-acities}$ with *neq(X, Y)*^{Act4}
Higher quality sequence:
    add-action: *unload-plane(O,Pl,Y)* to resolve *at-object(O, Y)*$_{Act1}$
    add-action: *load-plane(O,Pl,X)* to resolve *in(O, L)*$_{unload-plane}$
    add-action: *fly-plane(Pl,X,Y)* to resolve
        at-plane(Pl,Y))$_{unload-plane}$
    establish: *at-object(O, X)*$_{load-plane}$ with *at-object(O, X)*^{Act2}
    establish: *at-plane(Pl,X)*$_{fly-plane}$ with *at-plane(Pl, X)*^{Act3}
    establish: *neq(X,Y)*$_{fly-plane}$ with *neq(X, Y)*^{Act4}

Figure 3.16: Generalized planning decision sequences. Pre_{Act} denotes precondition Pre of Action *Act* and Eff^{Act} denotes effect Eff supplied by the action *Act*.

storing module identifies the following open-conditions and the effects that this planning decision sequence resolves:

open-conditions:{*at-object(O, Y)*$_{Act1}$}
effects:{*at-object(O, X)*^{Act2}, *at-plane(Pl, X)*^{Act3}, *neq(X, Y)*^{Act4}}.

The actions added by the better planning decision sequence form the subplan, *P*= {*load-plane(O, Pl, Y), fly-plane(Pl, Y, X), unload-plane(O, Pl, X)* }. The quality value of this subplan forms a part of the rule for applying this planning decision sequence. The quality value of a plan in the transportation domain is defined as $5\times time - money$, where $money$ and $time$ denote the amounts of the resources of time and money consumed by the plan. Computing these values for the subplan P and substituting these values in the quality formula yields: $Q = 5 * (20 + 20 + distance(Y, X)/100) - (15 + 15 + distance(Y, X)/5)$.

Putting all this together, the rule learned for the planning decision sequence associated with the higher-quality plan is shown in Figure 3.17. Similarly, by examining the planning decision sequence associated with the lower quality plan, PIP learns the rule shown in Figure 3.18. These rules specify that

open-conditions: {*at-object(O, Y)*$_{Act1}$}
effects: {*at-object(O, X)*^{Act2}, *at-plane(Pl, X)*^{Act3},
neq(X, Y)^{Act4}}
quality: `170 - 3 * distance(Y, X)/200.`
trace: *add-action:* *unload-plane(O,Pl,Y)* to resolve
at-object(O, Y)$_{Act1}$
add-action: *load-plane(O,Pl,X)* to resolve
in(O,L)$_{unload-plane}$
add-action: *fly-plane(Pl,X,Y)* to resolve
at-plane(Pl,Y))$_{unload-plane}$
establish: *at-object(O, X)* with *at-object(O, X)*^{Act2}
establish: *at-plane(Pl,X)* with *at-plane(Pl, X)*^{Act3}
establish: *neq(X,Y)* with *neq(X, Y)*^{Act4}

Figure 3.17: Search Control Rule 1: The rule formed by PIP for the higher quality decision sequence shown in Figure 3.15.

open-conditions: {*at-object(O, Y)*$_{Act1}$}
effects:{*at-object(O,X)*^{Act2}, *at-truck(Tr, X)*^{Act3},
neq(X, Y)^{Act4}}
quality: `50 - 2 * distance(Y, X)/25.`
trace: *add-action:* *unload-truck(O,Tr,Y)* to resolve
at-object(O, Y)$_{Act1}$
add-action: *load-truck(O,Tr,X)* to resolve
in(O,L)$_{unload-truck}$
add-action: *drive-truck-acities(Tr,X,Y)* to resolve
at-truck(Tr,Y))$_{unload-truck}$
establish: *at-object(O, X)* with *at-object(O, X)*^{Act2}
establish: *at-truck(Tr,X)* with *at-truck(Tr, X)*^{Act3}
establish: *neq(X,Y)* with *neq(X, Y)*^{Act4}

Figure 3.18: Search Control Rule 2: The rule formed by PIP for the lower quality decision sequence shown in Figure 3.15.

Initial-state: {at-object(o1, ap1), at-truck(tr1, po1), at-truck(tr2, ap2))
at-plane(pl1, ap1), same-city(ap1,po1),same-city(po1,ap1),
same-city(ap2,po2),same-city(po2,ap2), position(ap1, 0),
position(po1, 13), position(ap2, 221), position(po2, 230),
money(100), time(0)}

Goal: {at-object(o1, ap2)}

Figure 3.19: Problem 2: A Transportation planning problem.

the planning decisions specified in the `trace` field of the rule can resolve the goals/subgoals specified in the `open-conditions` field of the rule if all the members of the `effects` field of the rule are present in the partial plan's effect-set (i.e., the set E defined on page 43). The `quality` field of the rule specifies the effect on quality of the complete plan that resolving the flaws using the planning decisions (specified in the `trace` field of the rule) will have.

Retrieving the rules

Rules such as these are consulted by POP to produce a plan for similar subsequent problems. When refining a partial plan P, POP searches its rule library to find a rule whose open-conditions and effects are subsets of P's open condition set C_P and effect set E_P respectively. If more than one such rule is available, then the rule that has the largest precondition set (i.e., it resolves the largest number of preconditions) is selected. If more than one such rule is available, then POP uses the rule whose `quality` field has the highest value when evaluated in context of P.

Example 4.2: To see an illustration of rule-retrieval in PIP, suppose that after learning Search Control Rule 1 and Search Control Rule 2 (displayed in Figures 3.17 and 3.18), PIP is presented Problem 2 (the transportation problem shown in Figure 3.19).

PIP calls POP to solve this problem. POP's first step (Step 1 of the POP algorithm shown in Figure 3.4) is to initialize the partial plan $P =< A_P, O_P, L_P, E_P, C_P >$ as follows:

action-set $A_P \longleftarrow \{start,\ end\}$, ordering constraints set $O_P \longleftarrow \{start \succ end\}$,

causal-link set $L_P \longleftarrow \{\ \}$,

effect-set $E_P \longleftarrow \{$*at-object(o1, ap1)*$_{start}$, *at-truck(tr1, po1)*$_{start}$, *at-plane(pl, ap1)*$_{start}$, *at-plane(tr2, ap2)*$_{start}\}$, and

open-condition set $C_P \longleftarrow \{$*at-object(o1,ap2)*$_{end}\}$.

POP's next step (Step 2 of POP algorithm shown in Figure 3.4) is to call *refine*. Since the partial plan at this point contains a flaw, *refine* calls *retrieve* to see if a rule matches the partial plan P. Since the precondition and effect sets of both Rule 1 and Rule 2 (shown in Figure 3.17 and Figure 3.18) are subsets of P's precondition and effect-set, *retrieve* compares the quality values of the two rules computed in the context of the current partial plan to see which planning decision sequence promises to lead to a better quality plan. Since the quality value of Rule 1 ($170 - 3 * 221/200 = 166$) is higher than the quality value of Rule 2 ($50 - 2 * 221/200 = 32$), Rule 1 is selected for retrieval. The `trace` part of the rule containing the planning decisions,

```
{add-action(unload-plane(o2,pl1,ap2)),
 add-action(load-plane(o2,pl1,ap1)),
 add-action(fly-plane(pl1,ap1,ap2)),
 establish(at-object(o2, ap1)),
 establish(at-plane(pl1,ap1)),
 establish(neq(ap1,ap2))}.
```

is returned and sent to the replay procedure.

The replay procedure applies these planning decisions to the partial plan P to refine it. Following is the final plan produced by PIP for this problem:

```
{load-plane(o1, pl1, ap1),
 fly-plane(pl1, ap1, ap2),
 unload-plane(o1, pl1, ap2)}.
```

Refining PIP's Knowledge

A retrieved rule is guaranteed to guide the planner towards generating a higher-quality plan unless the partial plan has some yet unseen open-conditions that *negatively interact* with the preconditions in the antecedent of the rule. A negative interaction occurs if the application of a rule leads to a lower quality plan than the planner would have produced, had the rule not been used. PIP detects such cases during training when application of a rule leads to a lower quality plan than the model plan. When that happens, PIP learns a more specific rule.

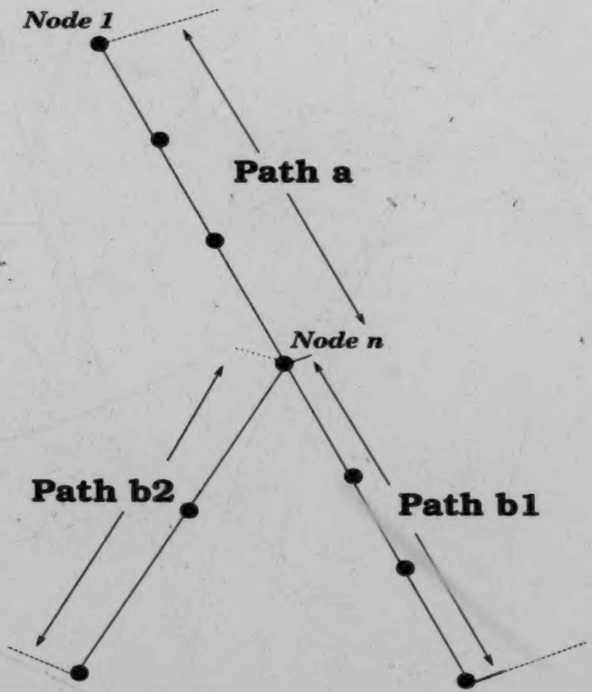


Figure 3.20: A conflicting choice point where application of a rule leads to a lower quality planning path (Path $a + b1$).

Suppose Path $a + b1$ in Figure 3.20 is the path followed by the system's planner because it was the path suggested by a retrieved rule R. Further suppose that Path $a + b1$ leads to a lower quality plan. This prompts PIP to identify a conflicting choice point that lies on a replayed node n. PIP learns a rule as usual for this conflicting choice point which would allow it to follow path $b2$ whenever it is at a node similar to node n. But this rule alone would not ensure the production of a better quality plan for similar problems. Even if the exact same problem is subsequently presented to PIP, it would never get to node n because at node 1 it would retrieve R and produce the lower quality decision again. This means that a rule must be learned that would apply in node 1. PIP does this by forming a rule (using the rule formation process discussed earlier) from the planning decisions that fall on the path $a + b2$. This rule is then added to PIP's rule base.

3.3 Summary

This chapter presents PIP's knowledge representation scheme. It also describes PIP's architecture and algorithms. In order to learn quality improving rules, PIP compares its planning episode with a better quality model planning episode. In apprenticeship situations where a user is only able to provide a model solution, PIP has to make a hypothesis about the model planning episode. Then it compares two planning episodes identifying the crucial planning decisions that are responsible for the difference in the overall qualities of the plans that resulted from the two episodes. PIP's learning opportunities are the conflicting choice points— these are the nodes in the search-tree for a problem where a flaw can be removed by applying two different planning decisions if these planning decisions lead to plans of different quality. The end product of this analysis is the identification of a set of flaws and for each of these flaws two different planning decision sequences are identified, both of which solve that flaw. This analysis is then stored in the form of the rule for each planning decision sequence. The idea is that learning this rule will help the planner decide which planning decisions to apply next time it is faced with a similar planning situation.

Chapter 4

PIP-rewrite

The motivation behind the work reported in this chapter was to study the benefits and costs of using the result of the analysis done in Step 3 of PIP's algorithm (shown in Figure 3.2) to formulate *plan-rewrite rules* (as opposed to the search control rules). Recall that current approaches to plan quality improvement via rewrite rules depend on hand-coded rewrite rules. While these approaches show the promise of rewrite rules for improving both planning efficiency and plan quality, they are impractical for most practical planning problems because of the difficulties involved in manually deriving and encoding the rewrite heuristics. The system (called PIP-rewrite) presented in this chapter learns plan-rewrite rules automatically and uses them to produce (presumably) better quality plans. PIP-rewrite follows the standard PIP algorithm described in the last chapter for the first three steps. In Step 4, PIP-rewrite selects all the relevant actions added by the planning decision sequences identified by ISL in Step 3 as relevant and stores these actions as a rewrite rule which essentially says "replace the lower quality actions with the high quality actions." Recall the two search control rules learned by PIP for Problem 1 (originally shown in Figures 3.17 and 3.18 and reproduced in Figure 4.1). PIP-rewrite identifies and stores the equivalent information as Rewrite Rule 1 shown in Figure 4.2.

This rule can then be used by PIP-rewrite *after* a complete plan for a similar subsequent problem has been generated by its default planner to rewrite it into a higher quality plan. The first part of this chapter presents details of how the

open-conditions: { *at-object(O, Y)*$_{Act1}$ }
effects: { *at-object(O,X)*^{Act2}, *at-plane(Pl, X)*^{Act3}, *neq(X,Y)*^{Act4}}.
quality: 170 - 3 * distance(Y, X)/200.
trace: *add-action:* *unload-plane(O,Pl,Y)* to resolve
at-object(O, Y)$_{Act1}$
add-action: *load-plane(O,Pl,X)* to resolve *in(O, L)*$_{unload\text{-}plane}$
add-action: *fly-plane(Pl,X,Y)* to resolve
at-plane(Pl,Y))$_{unload\text{-}plane}$
establish: *at-object(O, X)* with *at-object(O, X)*^{Act2}
establish: *at-plane(Pl,X)* with *at-plane(Pl, X)*^{Act3}
establish: *neq(X,Y)* with *neq(X, Y)*^{Act4}

open-conditions: { *at-object(O, Y)*$_{Act1}$ }
effects: { *at-object(O,X)*^{Act2}, *at-truck(Tr, X)*^{Act3}, *neq(X,Y)*^{Act4}}.
quality: 50 - 2 * distance(Y, X)/25.
trace: *add-action:* *unload-truck(O,Tr,Y)* to resolve
at-object(O, Y)$_{Act1}$
add-action: *load-truck(O,Tr,X)* to resolve *in(O, L)*$_{unload\text{-}truck}$
add-action: *drive-truck-acities(Tr,X,Y)* to resolve
at-truck(Tr,Y))$_{unload\text{-}truck}$
establish: *at-object(O, X)* with *at-object(O, X)*^{Act2}
establish: *at-truck(Tr,X)* with *at-truck(Tr, X)*^{Act3}
establish: *neq(X,Y)* with *neq(X, Y)*^{Act4}

Figure 4.1: Search Control Rule 1 and Search Control Rule 2, reproduced from Figures 3.17 and 3.18.

PIP framework presented in the last chapter can also be used for learning and using plan rewrite rules to improve both planning efficiency and plan quality. The second part evaluates the tradeoffs involved in employing rewrite versus search control rules in the PIP framework. These empirical investigations address the question: "Is it better to store the output of the PIP's learning module (i.e., ISL) as rewrite rules or as search control rules?" This matter is addressed by running both PIP and PIP-rewrite on a number of benchmark planning domains, measuring dependent variables such as plan quality and planning efficiency, and analyzing the results.

replace:
   actions: {load-truck(O,T,X),drive-truck-acities(T,X,Y),
               unload-truck(O,T,Y)}
   causal-links: {
     load-truck(O,T,X) $\xrightarrow{in-truck(O,T)}$ unload-truck(O,T,Y),
     drive-truck-acities(T,X,Y) $\xrightarrow{at-truck(T,Y)}$ unload-truck(O,T,Y)}
with:
   actions: {load-plane(O,L,X),fly-plane(L,X,Y),unload-plane(O,L,Y)}

Figure 4.2: Rewrite Rule 1: Learned by PIP-rewrite for the transportation problem shown in Figure 3.3.

4.1 PIP-rewrite's Architecture and Algorithm

PIP-rewrite has four main components of the PIP architecture (shown earlier in Figure 3.1) and follows PIP's high level algorithm (presented earlier in Figure 3.2). The learning algorithm used by PIP-rewrite is similar to that of PIP. The major difference is in the way the information returned by ISL is stored by PIP-rewrite. The following sections provide detailed algorithms for each of PIP-rewrite's components.

4.1.1 The Planning Component

Since PIP-rewrite does not learn any search-control rules, it does not use PIP's planner. PIP-rewrite uses a speed-up partial order planning algorithm called DerPOP to efficiently produce its initial plans. DerPOP is a Prolog version of the case-based partial order planner DerSNLP [IK97].

```
DerPOP (Init-state, Goals, Action-schemas)
    1- If retrieve(Init-state, Goals, Previous-case) then
        1.1- replay(Previous-case)
    2- else
        2.1- Planning-trace ⟵ POP(Init-state, Goals)
        2.2- store(Planning-trace)
```

Figure 4.3: DerPOP's planning algorithm.

As shown in Figure 4.3, DerPOP's first step is to see if goals and relevant initial conditions of a previously-cached planning trace are subsets of the goals and initial conditions of the current problem. If so it retrieves the planning trace. The retrieved case is then used by DerPOP to guide it to generate a plan for the current problem. If no previous case is available, then DerPOP plans from the first principles using POP. Given a planning problem, DerPOP produces a plan and a planning trace for that problem which constitute the default planning episode. The planning trace is one input to PIP-rewrite's learning component. The second input to the learning component, the model planning episode, is generated by PIP's standard model plan generator (described earlier in Section 3.2.2).

4.1.2 The Analytic Learning Component

Given the system's default plan and the model plan, the problem for PIP-rewrite's learning component is to identify subplan(s) of the default plan that can be *replaced* by subplan(s) of the model plan. Ambite [AK97] shows that a subplan s_1 of a plan P can be replaced by a subplan s_2 resulting in a plan P' iff:

1. *preconditions*$(s_2) \subseteq$ *effects*$(s_2 \cup P - s_1)$, and
2. *useful-effects*$(s_1) \subseteq$ *effects*$(s_2 \cup P - s_1)$, and
3. an ordering of actions exists such that P' is a viable plan,

where useful effects of a subplan s of a plan P are defined as the predicates present in the causal-links whose producer is in S and whose consumer is in $P - S$. Condition 1 is necessary to ensure that all of s_2's preconditions can be satisfied. Condition 2 is necessary to ensure that all the preconditions of $P - s_1$ that used to be supplied by s_1 can still be satisfied.

A naive algorithm for learning plan-rewrite rules then would be to compare all subplans s_{1_i} of the default plan with all subplans s_{2_j} of the model plan to identify which s_{1_i} can be rewritten by which s_{2_j}. Clearly, the computational complexity of this problem is exponential in the number of actions in both the default plan and the model plan. This makes it computationally infeasible for any large problem.

PIP-rewrite uses a heuristic approach which is more efficient in practice but provides no guarantee that the replacing subplan can replace the to-be-replaced subplan in the default plan. The idea is to focus on the problem subgoals and find subplan(s) in the default plan that are *equivalent* with the subplan(s) in the model plan. Two subplans are considered equivalent if they both solve the same subgoal. PIP-rewrite uses the ISL algorithm (described earlier in Section 3.2.3) to compute the to-be-replaced and the replacing subplans. However, PIP-rewrite supplies ISL with a completely instantiated default planning trace (instead of an uninstantiated trace as is done in PIP) to transform all *establish—add-action*, *add-action—establish* and *establish—establish* type conflicting choice points into *add-action—add-action* type conflicting choice points.

The reason for this modification is this. Sometimes the way in which the two refinement paths out of a conflicting choice point differ is that the worse plan-refinement path uses only establishment decisions (i.e., decisions to use existing actions) to resolve the open condition flaws, while the higher quality path resolves them using some add-action decisions. This can lead to rewrite-rules of the sort:

```
replace:
     actions: {}
     causal-links: {}
with:
     actions: {drive-truck(T,X,Y)}.
```

Note that the effect of such a rule is to simply add actions to a plan under any conditions.

Instantiating the planning trace transforms all the *establish*-type conflicting choice points into *add-action*-type conflicting choice points. This way the only conflicting choice points identified by ISL are *add-action*-type conflicting choice points. I illustrate this with the help of the Transportation example shown in Figure 4.4. Figure 4.5 shows the uninstantiated planning trace returned by DerPOP. This trace is called uninstantiated because during DerPOP's derivation of the plan, values of some variables are uninstantiated. For

Initial-state: {at-object(letter, edm-ap), at-plane(plane1, edm-ap),
at-plane(plane2, cal-ap), neq(ap1,ap2), neq(ap2,ap1)}
Goal: {at-object(letter, cal-ap)}

System's Default Plan	**Model Plan**
load-plane(letter, plane1, edm-ap)	**fly-plane(plane2, cal-ap, edm-ap)**
fly-plane(plane1, edm-ap, cal-ap)	**load-plane(letter, plane2, edm-ap)**
unload-plane(letter, plane1, cal-ap)	**fly-plane(plane2, edm-ap, cal-ap)**
	unload-plane(letter, plane2, cal-ap)

Figure 4.4: A Transportation problem. A letter is at Edmonton Airport (*edm-ap*) in the initial state and the goal is to get it to Calgary Airport (*cal-ap*). Default planer uses *plane*1 for transporting the object and the model plan uses *plane*2 to fly the object.

instance, the variable *Pl* is uninstantiated in Nodes 2-4. The variable denoting the location from where to fly the plane (*From2*) also remains uninstantiated until the precondition *at-plane(Pl, From2)*$_{fly\text{-}plane}$ is established with the effect *at-plane(pl1, edm-ap)* present in the initial condition set.

Figure 4.6 shows the instantiated trace. This trace is called instantiated because all the variables have been replaced by constants with which they are eventually bound (later in the search). For instance, the variable *Pl1* has been replaced by the constant *pl1* and the variables *From1* and *From2* have been replaced by *edm-ap* and *cal-ap* respectively.

Figure 4.7 shows the conflicting choice point identified by ISL using the uninstantiated trace. The conflicting choice point in this case is at the decision point of resolving the open-condition *at-plane(Plane, AP)* which the default planner resolves by establishment with the condition *at-plane(plane1, edm-ap)*$_{start}$ present in the initial state. This is an *establish−add-action* conflicting choice point. However, when ISL is given the instantiated trace (shown in Figure 4.8) then the conflicting choice point moves up (in the search tree) to the resolution of the open-condition *at-object(letter, cal-ap)*. The default planner resolves it by adding the action *unload-plane(letter, plane1, cal-ap)* and the model planner resolves it by adding the action *unload-plane(letter, plane2,*

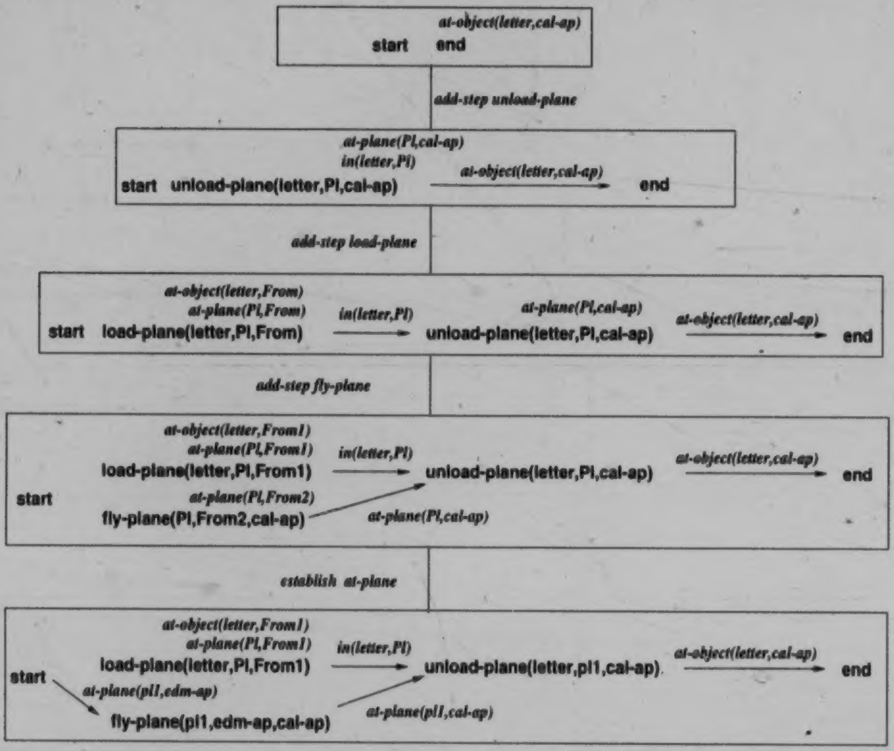

at-object(letter,cal-ap)
start
end
add-step unload-plane
at-plane(Pl,cal-ap)
in(letter,Pl)
at-object(letter,cal-ap)
start unload-plane(letter,Pl,cal-ap)
end
add-step load-plane
at-object(letter,From)
at-plane(Pl,From)
in(letter,Pl)
at-plane(Pl,cal-ap)
at-object(letter,cal-ap)
start load-plane(letter,Pl,From)
unload-plane(letter,Pl,cal-ap)
end
add-step fly-plane
at-object(letter,From1)
at-plane(Pl,From1)
in(letter,Pl)
load-plane(letter,Pl,From1)
unload-plane(letter,Pl,cal-ap)
at-object(letter,cal-ap)
end
start
at-plane(Pl,From2)
fly-plane(Pl,From2,cal-ap)
at-plane(Pl,cal-ap)
establish at-plane
at-object(letter,From1)
at-plane(Pl,From1)
in(letter,Pl)
load-plane(letter,Pl,From1)
unload-plane(letter,pl1,cal-ap)
at-object(letter,cal-ap)
end
start
at-plane(pl1,edm-ap)
fly-plane(pl1,edm-ap,cal-ap)
at-plane(pl1,cal-ap)

Figure 4.5: Uninstantiated planning trace for the default plan shown in Figure 4.4. Please note that only top part of the planning trace is shown for brevity.

cal-ap). Thus treating the two differently instantiated actions as two different actions allows PIP-rewrite to translate all the conflicting choice points involving *establishment* into *add-action–add-action* type conflicting choice points.

The output of ISL-rewrite is two planning decision sequences that resolve the same subgoal/goal.

4.1.3 The Rule library

Forming and Storing the rule

Given the two planning decision sequences, PIP-rewrite computes the actions added by each sequence to compute the two subplans that solve the same goal and stores that information in the form of a rule. The actions that are added by the worse plan's planning decision sequence become the subplan to be replaced and the actions that are added by the better plan's decision sequence become

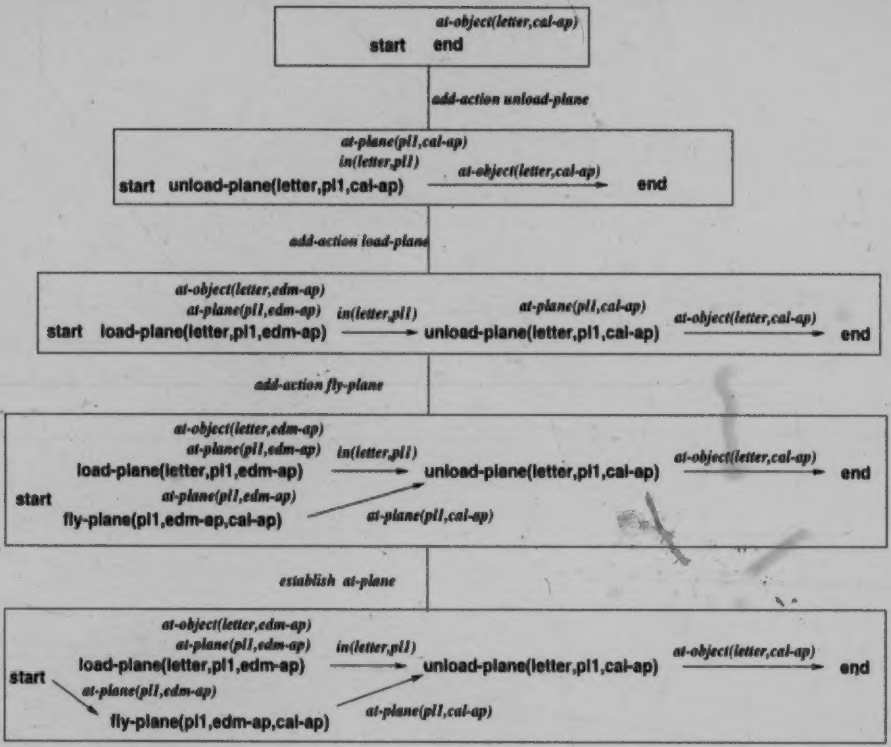

at-object(letter,cal-ap)
start
end
add-action unload-plane
at-plane(pl1,cal-ap)
in(letter,pl1)
start
unload-plane(letter,pl1,cal-ap)
at-object(letter,cal-ap)
end
add-action load-plane
at-object(letter,edm-ap)
at-plane(pl1,edm-ap)
start
load-plane(letter,pl1,edm-ap)
in(letter,pl1)
at-plane(pl1,cal-ap)
unload-plane(letter,pl1,cal-ap)
at-object(letter,cal-ap)
end
add-action fly-plane
at-object(letter,edm-ap)
at-plane(pl1,edm-ap)
load-plane(letter,pl1,edm-ap)
in(letter,pl1)
unload-plane(letter,pl1,cal-ap)
at-object(letter,cal-ap)
end
start
at-plane(pl1,edm-ap)
fly-plane(pl1,edm-ap,cal-ap)
at-plane(pl1,cal-ap)
establish at-plane
at-object(letter,edm-ap)
at-plane(pl1,edm-ap)
load-plane(letter,pl1,edm-ap)
in(letter,pl1)
unload-plane(letter,pl1,cal-ap)
at-object(letter,cal-ap)
end
start
at-plane(pl1,edm-ap)
fly-plane(pl1,edm-ap,cal-ap)
at-plane(pl1,cal-ap)

Figure 4.6: Instantiated planning trace for the default plan shown in Figure 4.4. Please note that only the top part of the planning trace is shown for brevity.

the replacing subplan. PIP-rewrite also identifies the causal links added by the worse planning decision sequence between the to-be-replaced actions as the to-be-replaced causal links. This information is then stored in the rule library as a rewrite rule.

Consider again the planning decision sequence shown in Figure 3.16. The actions added by the lower quality decision sequence are {*load-truck(O,T,X), drive-truck-acities(T,X,Y),unload-truck(O,T,Y)*}, and the causal links involving these actions added by the better decision sequence are {*load-truck(O,T,X)* $\xrightarrow{in-truck(O,T)}$ *unload-truck(O,T,Y), drive-truck-acities(T,X,Y)* $\xrightarrow{at-truck(T,Y)}$ *unload-truck(O,T,Y)*}. Similarly, the actions added by the higher quality decision sequence are {*load-plane(O,L,X),fly-plane(L,X,Y),unload-plane(O,L,Y)*}. PIP-rewrite stores this information as Rewrite Rule 1 shown in Figure 4.2.

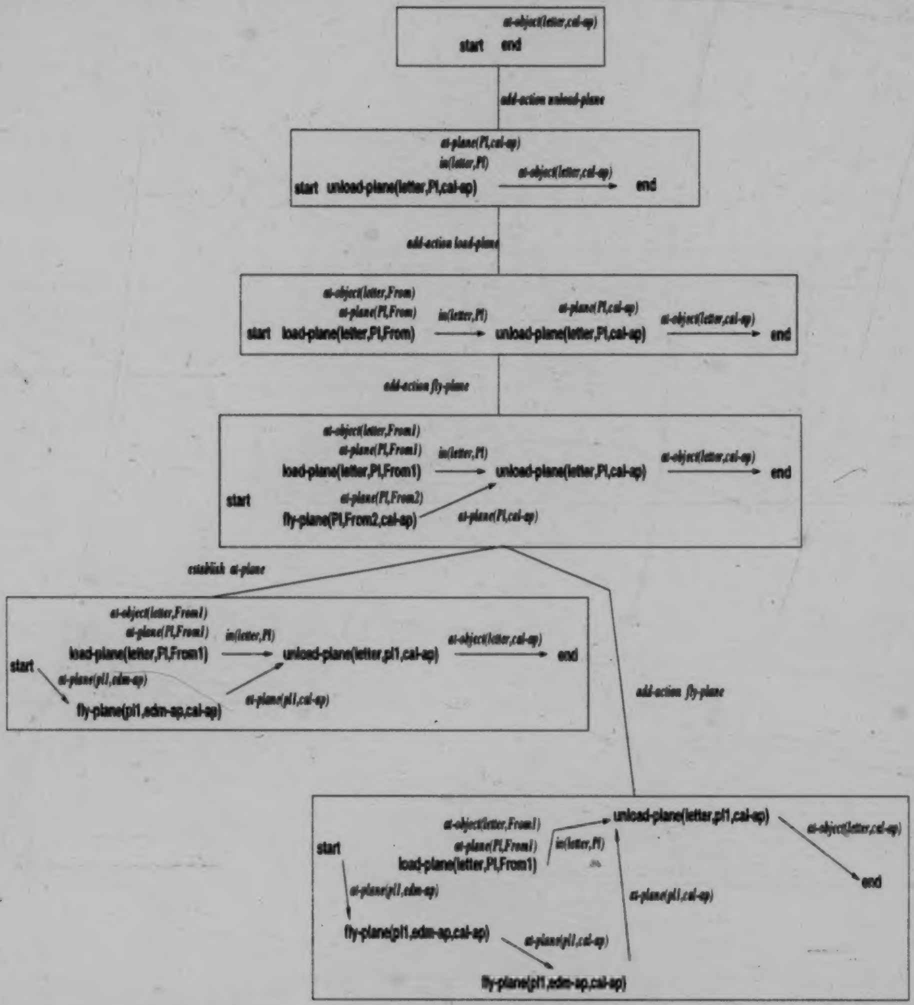

at-object(letter,cal-ap)
start
end
add-action unload-plane
at-plane(Pl,cal-ap)
in(letter,Pl)
at-object(letter,cal-ap)
start
unload-plane(letter,Pl,cal-ap)
end
add-action load-plane
at-object(letter,From)
at-plane(Pl,From)
in(letter,Pl)
at-plane(Pl,cal-ap)
at-object(letter,cal-ap)
start
load-plane(letter,Pl,From)
unload-plane(letter,Pl,cal-ap)
end
add-action fly-plane
at-object(letter,From1)
at-plane(Pl,From1)
in(letter,Pl)
at-object(letter,cal-ap)
load-plane(letter,Pl,From1)
unload-plane(letter,Pl,cal-ap)
end
start
at-plane(Pl,From2)
fly-plane(Pl,From2,cal-ap)
at-plane(Pl,cal-ap)
establish at-plane
at-object(letter,From1)
at-plane(Pl,From1)
in(letter,Pl)
at-object(letter,cal-ap)
load-plane(letter,Pl,From1)
unload-plane(letter,pl1,cal-ap)
end
start
at-plane(pl1,edm-ap)
fly-plane(pl1,edm-ap,cal-ap)
at-plane(pl1,cal-ap)
add-action fly-plane
unload-plane(letter,pl1,cal-ap)
at-object(letter,From1)
at-object(letter,cal-ap)
start
at-plane(Pl,From1)
in(letter,Pl)
load-plane(letter,Pl,From1)
end
at-plane(pl1,edm-ap)
at-plane(pl1,cal-ap)
fly-plane(pl1,edm-ap,cal-ap)
at-plane(pl1,cal-ap)
fly-plane(pl1,edm-ap,cal-ap)

Figure 4.7: Learning opportunities identified by ISL using the uninstantiated default trace shown in Figure 4.5.

Retrieving the rules

When given a problem to solve, PIP-rewrite's default planner DerPOP produces a complete plan P_i which includes the set of actions A_{P_i}, the set of casual links Cl_{P_i}, ordering constraints O_{P_i} and the set of effects E_{P_i}. PIP-rewrite's next step is to search its rule library to find a rule whose $Actions_{to\text{-}be\text{-}replaced}$ are a subset of A_{P_i} and whose causal link constraints $Cl_{to\text{-}be\text{-}replaced}$ are a subset of Cl_{P_i}. If any such rule

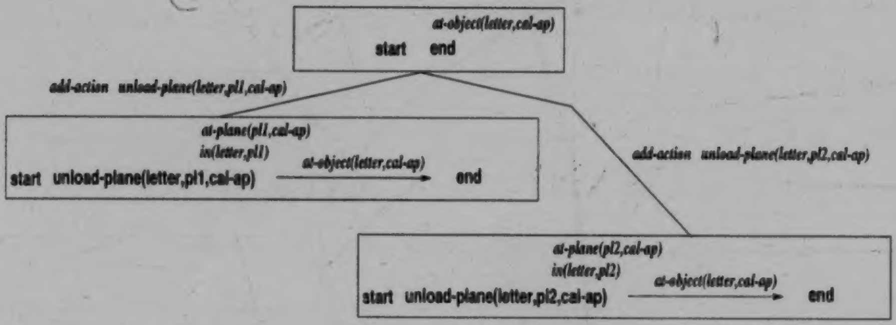

at-object(letter,cal-ap)
start end
add-action unload-plane(letter,pl1,cal-ap)
at-plane(pl1,cal-ap)
in(letter,pl1)
at-object(letter,cal-ap)
start unload-plane(letter,pl1,cal-ap) end
add-action unload-plane(letter,pl2,cal-ap)
at-plane(pl2,cal-ap)
in(letter,pl2)
at-object(letter,cal-ap)
start unload-plane(letter,pl2,cal-ap) end

Figure 4.8: Learning opportunities identified by ISL using the instantiated planning trace shown in Figure 4.6.

$R = (Actions_{to\text{-}be\text{-}replaced}, Cl_{to\text{-}be\text{-}replaced}, Actions_{replacing})$ is retrieved, then all the ordering constraints from O_{P_i} that involve an action from $Actions_{to\text{-}be\text{-}replaced}$ are deleted. It also deletes all causal links from $Cl_{to\text{-}be\text{-}replaced}$ whose producer is a member of $Actions_{to\text{-}be\text{-}replaced}$. All those conditions in the casual-links that have a producer in $Actions_{to\text{-}be\text{-}replaced}$ and a consumer in $P - Actions_{to\text{-}be\text{-}replaced}$ are added to the set of open conditions. The replacing action sequence is appended to the set of actions to obtain the new partial plan $P_j = (Acts, Effs, Open\text{-}conds, Cl, O)$, where

$$Acts = A_{P_i} - Actions_{to\text{-}be\text{-}replaced} \cup Actions_{replacing}$$

$$E = E_p - \{e \mid e \text{ is added by an action a } \in Actions_{to\text{-}be\text{-}replaced} \cup Actions_{replacing}\}$$

$$C = \{c^{a_2} \mid a_1 \xrightarrow{c} a_2 \in Cl_p, a_1 \in Actions_{to\text{-}be\text{-}replaced}, a_2 \in P_i - Actions_{to\text{-}be\text{-}replaced}\} \\ \cup \{c^a \mid a \in Actions_{replacing}\}$$

$$Cl = Cl_p - \{a_1 \xrightarrow{c} a_2 \mid a_1 \in Actions_{to\text{-}be\text{-}replaced} \cup Actions_{to\text{-}be\text{-}replaced}\}$$

$$O = O_p - \{a_1 \succ a_2 \mid a_1 \in Actions_{to\text{-}be\text{-}replaced} \cup Actions_{to\text{-}be\text{-}replaced}\}.$$

After applying a rule, the rewritten plan P_j can be rewritten again if any applicable rules exist or it can be refined to remove its flaws in order to turn it into a complete plan.

Example 5.1: To see an illustration of PIP-rewrite's rule retrieval and plan rewriting process, suppose that after learning Rewrite Rule 1 (shown in Figure

Initial-state: {at-object(o1, ap1), at-truck(tr1, po1), at-truck(tr2, ap2)}
at-plane(pl1, ap1), same-city(ap1,po1),same-city(po1,ap1),
same-city(ap2,po2),same-city(po2,ap2), position(ap1, 0),
position(po1, 13), position(ap2, 221), position(po2, 230),
money(100), time(0)}

Goal: {at-object(o1, ap2)}

Figure 4.9: Problem 2: A Transportation planning problem.

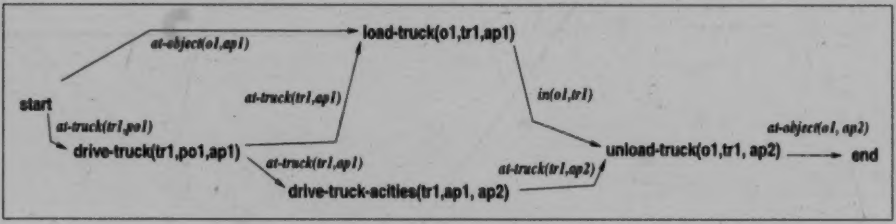

load-truck(o1,tr1,ap1)
at-object(o1,ap1)
at-truck(tr1,ap1)
in(o1,tr1)
start
at-truck(tr1,po1)
drive-truck(tr1,po1,ap1)
at-truck(tr1,ap1)
at-object(o1, ap2)
unload-truck(o1,tr1, ap2)
end
at-truck(tr1,ap2)
drive-truck-acities(tr1,ap1, ap2)

Figure 4.10: DerPOP's plan for the problem shown in Figure 3.19.

4.2) PIP-rewrite is given the problem originally presented in Figure 3.19 and reproduced in Figure 4.9. PIP-rewrite calls its default planner DerPOP to produce the plan shown in Figure 4.10 for this problem.

Since to-be-replaced actions and to-be-replaced causal links of Rewrite Rule 1 (shown in Figure 4.2) are subsets of PIP-rewrite's initial plan and its causal links, PIP-rewrite retrieves the rule shown in Figure 4.11.

The retrieved rule is then applied to PIP-rewrite's initial plan. This means deleting the to-be-replaced actions and to-be-replaced causal links (as specified

`replace:`

`actions:` {load-truck(o1,tr1,ap1),drive-truck-acities(tr1,ap1,ap2), unload-truck(o1,tr1,ap2)}

`causal-links:`

load-truck(o1,tr1,ap1) $\xrightarrow{in-truck(o1,tr1)}$ unload-truck(o1,tr1,ap2),

drive-truck-acities(tr1,ap1,ap2) $\xrightarrow{at-truck(tr1,ap2)}$ unload-truck(o1,tr1,ap2)}

`with:`

`actions:` load-plane(o1,Pl,ap1),fly-plane(Pl,ap1,ap2),

Figure 4.11: Rule retrieved by PIP-rewrite.

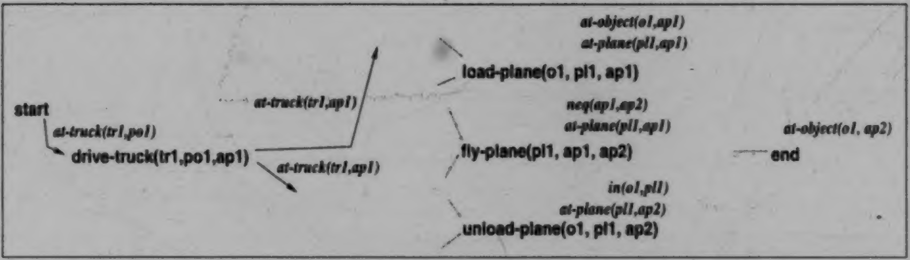

at-object(o1,ap1)
at-plane(pl1,ap1)
load-plane(o1, pl1, ap1)
start
at-truck(tr1,ap1)
at-truck(tr1,po1)
drive-truck(tr1,po1,ap1)
at-truck(tr1,ap1)
neq(ap1,ap2)
at-plane(pl1,ap1)
fly-plane(pl1, ap1, ap2)
at-object(o1, ap2)
end
in(o1,pl1)
at-plane(pl1,ap2)
unload-plane(o1, pl1, ap2)

Figure 4.12: The initial plan after the application of Rewrite Rule 1. Broken lines indicate the open conditions flaws introduced by rewriting i.e., preconditions of the actions that need to be satisfied.

by the retrieved rule shown in Figure 4.11) from the initial plan shown in Figure 4.10) and adding the replacing-actions (specified in the retrieved rule of Figure 4.11). For instance the causal-link *unload-truck(o1,tr1,ap2)* $\xrightarrow{at-object(o1,ap2)}$ *end* is also deleted because its supporting action *unload-truck(o1,tr1,ap2)* is a to-be-replaced action. Figure 4.12 displays the plan obtained by this deletion/addition process. Since there are no more applicable rules in PIP-rewrite's library, no more rewrites are possible.

However, application of the rewrite rule has turned a complete plan into an incomplete plan (shown in Figure 4.12) i.e., a plan that has some flaws in it. Figure 4.12 represents the open condition flaws by broken lines coming out of the actions that need these preconditions. The *refine* procedure shown in Figure 4.13 is then called to refine this partial plan. Note that this algorithm is similar to the *refine* procedure of the POP algorithm shown in Figure 3.4. The main difference is that in this algorithm, the only way to resolve open conditions is via establishment decisions. This makes the rewrite algorithm less flexible but simpler (and hence more efficient) than the partial-order planning algorithm. It also means that not all incompleted plans obtained by applying a rewrite rule to them can be resolved by the *refine* algorithm (e.g., those incomplete plans that have some open conditions that can only be resolved by the add-action planning decisions).

Applying the *refine* procedure of Figure 4.13 to the incomplete plan shown in Figure 4.12 results in the following complete plan, which has higher quality

```
refine (P, Ptr)
    If not flaw(P) then
        return success
    else
        if unsafe-links(P, Threats) then
            if resolve-threats(Threats, P, Ptr) then
                    {same as POP's resolve-threats shown in Figure 3.4}
                (P, Ptr) ⟵ resolve-threats(Threats, P, Ptr)
                return refine(P, Ptr)
            else
                fail
        if ∃ $c^{a_i} \in C_P$ then
            if resolve-an-open-cond($c^{a_i}$, P) then
                (P, Ptr) ⟵ resolve-an-open-cond($c^{a_i}$, P, Ptr)
                return refine(P, Ptr)
            else
                fail

resolve-an-open-cond ($c^{a_i}$, P, Ptr)
    -If ∃ an action $a_j \in A_P$ that adds c then
        - {establish}
            - $L_P \longleftarrow L_P \cup \{a_j \xrightarrow{c} a_i\}$
            - $O_P \longleftarrow O_P \cup \{a_j \succ a_i\}$
            - Ptr ⟵ Ptr $\cup \{a_j \xrightarrow{c} a_i\}$
            return (P, Ptr)
    else
        fail
```

Figure 4.13: The refine algorithm of PIP-rewrite

than system's initial-plan:

```
{drive-truck(tr1, po1, ap1),
load-plane(o1, pl1, ap1),
fly-plane(pl1, ap1, ap2),
unload-plane(o1, pl1, ap2)}.
```

Notice however that PIP-rewrite is unable to produce an optimal quality plan for Problem 2 after having been trained on Problem 1. Recall that PIP was able to produce the optimal quality plan using the search control rule it learned from Problem 1. I will return to this issue in Section 4.2.5.

Parameters for plan rewriting. If some of the rewrite rules in the rule library *undo* each other's rewriting, then the recursive rewrite process can go on forever. Therefore, a limit has to be placed on the number of rewrites. Currently, PIP-rewrite only makes two rewrites to a plan. Another variable in a plan-rewrite system is the number of ways the initial plan can be rewritten in each rewrite-step. The reason is that a number of rules may be applicable to a plan. Application of each of these rules may lead to a number of different rewritten plan(s) of different quality. This number can be as large as the number of ways of applying (i.e., instantiating) all the applicable rewrite rules. The benefit of applying all rewrite rules is that it allows evaluation of the entire neighborhood and hence the best quality plan can be obtained. However, searching the entire neighborhood can be inefficient. If we restrict the ways of rewriting a plan to the first feasible way of rewriting, then the rewrite algorithm becomes efficient. The drawback is that we are not making use of all the learned knowledge. A compromise between these two extremes is to use a local search strategy such as hill-climbing. For the experiments reported in the next two chapters, two versions of PIP-rewrite were implemented: PIP-rewrite-best, which explores all ways of rewriting and PIP-rewrite-first, which stops after computing the first rewritten plan. PIP-rewrite-first returns the rewritten plan only if it has higher quality than the system's initial plan. If the initial plan has a higher quality than the rewritten plan then the initial plan is returned by PIP-rewrite-first.

PIP-rewrite-first is the best that PIP-rewrite can do in terms of planning efficiency and PIP-rewrite-best is the best that PIP-rewrite can do in terms of improving plan quality. This allowed us to compare the best performance of PIP-rewrite with that of PIP.

4.2 Comparison of Rewrite and Search Control Rules

Clearly the best a planning by rewriting system can do in terms of planning efficiency is as good as its base planner that produces the initial plan, while a search control system can potentially be more efficient than its base planner. The only reason why planning by rewriting is argued to be able to improve both planning efficiency and plan quality is that such system can employ a speed-up planner such as DerPOP as its base planner while a search control system cannot. Given such a set up, it is not clear as to which technique (i.e., search control rules or rewrite rules) is a better strategy for storing the knowledge learned by the PIP's analytic learning process. This section presents an empirical comparison of the two techniques to see what improvements in planning efficiency and quality are obtained by the two techniques. First the experimental methodology is described, then the problem domains are discussed, and finally the experimental results are presented.

4.2.1 Methodology

The experimental methodology of cross validation was used for the experiments reported here and in the next chapter. A problem set containing 120 unique problems was randomly generated and 20, 30-, 40-, and 60-fold cross-validations were performed. The cross-validation procedure for an x-item ($x = 20, 30, 45, 60$) training set (or ***x*-problem set** as I will refer to them in the rest of the document) involves generation of $\frac{120}{x}$ unique problem sets each consisting of x training items and $\frac{120}{x}$ testing items. This ensures that after all the $\frac{120}{x}$ runs, each of the total of 120 problems has appeared $\frac{120}{x}$ times as a test problem. For instance, in the case of 20-problem set, the 120 problem set is divided into $\frac{120}{20} = 6$ data sets. Each of these data sets contains 20 training problems and 100 testing problems. PIP is then run on each of these six data sets. Similarly, there are $\frac{120}{30} = 4$ cross validation runs in case of the 30-problem set, $\frac{120}{40} = 3$ cross validation runs for the 40-problem set, and $\frac{120}{60} = 2$ cross validation runs in case of the 60-problem set.

Metrics of Interest

Performance of a planning and learning system can be measured along a number of dimensions. Most significant among these are the planning efficiency and plan quality. Other factors include the utility of the learned knowledge and the scalability of the techniques.

Plan Quality. Average plan length is the metric that is used by most existing planning systems to measure plan quality, mainly because they define plan quality as plan length. A measure equivalent to that in a system that has more complex representation of plan quality would be the average quality value of all the plans produced by the system for the test problems. This statistic provides some measure of the improvement in quality value within a domain but does not allow comparisons across different domains because the quality values between the two domains could differ widely.

An alternative statistic for measuring plan quality is the percentage of the plans produced by the planner that are of optimal quality. If P_i is the plan produced by a planner for the ith testing problem, N is the number of testing problems, and M_{P_i} denotes the model plan for this problem then

$$Q_1 = \frac{\sum_{i=1}^{N} equal(quality(P_i),\ quality(M_{P_i}))}{N}$$

$$equal(X,Y) = \begin{pmatrix} 1 \text{ if } X = Y \\ 0 \text{ otherwise} \end{pmatrix}.$$

This statistic provides some measure of the improvements in a planning system's performance on quality but it ignores the improvements that occur when the system produces a better solution (than it would have produced without any learning) but not an optimal quality solution. This is a problem in domains where each problem may have, on average, multiple solutions of different quality.

One solution to deal with this problem is to compute the average difference in the quality value of the plan produced by the system and the quality value of the optimal quality plan. This metric can be computed as:

$$Q_2 = \frac{\sum_{i=1}^{N} quality(M_{P_i}) - quality(P_i)}{N}.$$

However, this statistic also does not allow comparisons across two different domains because the quality values between the two domains could differ widely. A solution to this problem is to normalize the average plan quality. The normalization factor used in the results reported here (and in the next chapter) is the average distance of the plans produced by the non-learning planner from the optimal quality plans. If $P_{l1}, \ldots, P_{lN}$ denote the plans produced by the planner after having learned l examples (i.e., $P_{01}, \ldots P_{0N}$ denote the plans produced by a planner without any learning) then the new plan quality metric is given as[1]:

$$Q_3 = \frac{\sum_{i=1}^{N} quality(M_{P_{li}}) - quality(P_{li})}{\sum_{i=1}^{N} quality(M_{P_{0i}}) - quality(P_{0i})} \qquad (4.1)$$

where N denotes the number of test problems.

In the experiments reported in this chapter and the next chapter, the value of the metric Q_3 was computed for each of the $\frac{120}{x}$ cross validation runs for an x-problem set. These values were then added to compute the sum of all Q_3 values which was then divided by the number of cross validation runs (i.e., $\frac{120}{x}$) to get the mean value (m_Q) of the plan quality metric

$$m_Q = \frac{\sum_{i=1}^{\frac{120}{x}} Q_{3i}}{\frac{120}{x}} \qquad (4.2)$$

where Q_{3i} is the value of the metric Q_3 measured for the ith cross-validation run. For instance, in the 20-problem set case, six values of the plan quality metric (i.e., $Q_{31}, Q_{32}, Q_{33}, Q_{34}, Q_{35}, Q_{36}$) were calculated (corresponding to the six cross validations runs), their sum computed and divided by six to compute the mean value of the plan quality metric:

---

[1] The value of this metric cannot be computed when the denominator (i.e., average distance of the plans produced by the non-learning planner from the optimal quality plans) is zero. This only happens when the planner produces model plans without any learning. This situation never arises in the experiments reported here.

$$m_Q = \frac{Q_{31} + Q_{32} + Q_{33} + Q_{34} + Q_{35} + Q_{36}}{6}.$$

If learning is effective in guiding the planner towards good planning paths, then the normalized average plan quality distance (and hence the mean value of the plan quality metric) should decrease as learning progresses. And if the method has general applicability, then this decrease should occur in many different domains.

Planning Efficiency. A number of statistics are used for measuring planning efficiency of planning and learning systems. These include the CPU time taken to compute a plan (including the rule retrieval time), CPU time taken to generate a plan not counting the rule retrieval time, and the number of search nodes the planner needs to expand to generate a plan. However, it is difficult to draw any conclusions by comparing the planning times of two algorithms because of the differences in the compilers, platforms, and implementation techniques.

Here (and in the next chapter), I use the number of partial plans (denoted by $NumPP$) that PIP's planner needs to expand to generate a solution for a problem to measure planning efficiency. If learning is effective in biasing the planner towards good planning paths and away from bad planning paths, then the average number of nodes needed to be expanded should decrease as the learner is exposed to more training examples.

Similar to the case of the plan quality metric, the average number of partial plans generated per problem (i.e., $NumPP$) was counted for each of the $\frac{120}{x}$ cross validation runs for an x-problem set. These values were added to compute the sum $\sum_{i=1}^{\frac{120}{x}} NumPP$ which was then divided by the number of cross validation runs to compute the mean

$$m_{NumPP} = \frac{\sum_{i=1}^{\frac{120}{x}} NumPP_i}{\frac{120}{x}} \tag{4.3}$$

where $NumPP_i$ is the average number of partial plans per problem measured for the ith cross-validation run.

Other metrics of interest Learning search control rules seems like an attractive strategy because such rules can potentially improve the performance of a planner by biasing it towards promising planning paths. Management of these rules, however, has a certain cost associated with it that also has to be considered when evaluating the search control rules. This is the cost of retrieving and evaluating the control rules at each choice point during the search. Because of this cost, it is desirable to learn only those rules that are useful towards the production of good solutions. Descriptive statistics that can provide some measure of the utility of the learned rules include: the size of the rule library (*NumRules*), the number of the rules that were actually used in the construction of a plan (*NumUseful*), and the number of rules that needed to be revised (*NumRevised*).

The value of each of these metrics (i.e., *NumRules, NumUseful*, and *NumRevised*) was calculated for each of the $\frac{120}{x}$ cross validation runs of the x-problem set. These values were then added to get a sum $\sum_{i=1}^{\frac{120}{x}} Num_i$, which was then divided by the number of cross validation runs to get the mean value of each metric

$$m_{Num} = \frac{\sum_{i=1}^{\frac{120}{x}} Num_i}{\frac{120}{x}}. \tag{4.4}$$

4.2.2 Domain Descriptions

The purpose of empirical experiments reported here was to see if PIP and PIP-rewrite can learn to improve plan quality in a diverse set of "naturally inspired" domains. Three domains were selected for the experiments: Softbot [Wil96], Transportation domain [UE98], and Minton's manufacturing process planning domain [Min89].

The Transportation Domain

The transportation domain was derived from Veloso's logistics domain [Vel94]. The original logistics domain modeled a package delivery domain. Each city in that domain contained two locations (airports and post-offices) and a truck. Each location may also contain some packages and each airport may have some airplanes. The goal is to transport the packages from one location to another location. Trucks are used to transport packages within the same city, and planes are used to transport packages between different cities. I extended the original logistics domain by adding the action *move-truck-acities(Truck, From, To)* to provide an alternative means of moving between the cities and by adding resources of time and money and a quality function. Action descriptions were also modified so that metric effects of each action specify how the action changes the amount of money and the time in the world. For instance, the time-taken and the cost of *move-truck(Truck, From, To)* is defined as a function of the *distance* between locations *From* and *To*. Plan quality is defined as *quality(time,money) = 5 * time − money*. PR-STRIPS encoding of Transportation domain is shown in Appendix A.

In the transportation domain, the initial-state is described by propositional as well as by metric attributes (representing the initial values of the resources of money and time). The places (i.e., airports AP and ports PO) have *positions*. Problems are produced by generating random initial states and goals. Place positions are also assigned random values. If places are in the *same-city*, distances between them are generated to be less than a *short-distance*, where distance between the places *From* and *To* is calculated as *distance(From, To) = abs(position(From) - position(To))*, where *position(From)* and *position(To)* are real numbers that denote the position of the place *From* and the place *To* respectively.

The Softbot Domain

The Softbot domain was developed by Williamson [Wil96] and inspired by the Rodney Softbot Project at the University of Washington [EW94]. It models a simple software agent using various Internet-based resources for information gathering. The agent can use operators such as *finger* or *netfind* to solve goals such as knowing a person's phone number or email address. The quality variables of interest are time, money, help and bother (how much would it bother another agent if this action was taken by the planning agent) and quality of a plan is simply the sum of all the resources consumed by the plan. A PR-STRIPS encoding of Softbot domain is shown in Appendix B.

Manufacturing Process Planning Domain

The task in the manufacturing process planning domain [Min89] is to find a plan to manufacture a set of parts. The domain contains a variety of machines, such as a lathe, punch, spray painter, welder, etc., for a total of ten machining operations. The operator specifications are shown in Appendix C. The features of each part are described by a set of predicates such as temperature, painted, has-hole, etc. These features are changed by the operators. Other predicates that are not added by any action such as has-clamp, is-drillable, etc., are true in the initial state.

Each action is assigned a cost metric representing the cost of that action. Cost of a plan is the sum of the costs of its actions. Quality of a plan is defined as 1/cost i.e., the lower the cost of a plan, the higher its quality.

4.2.3 Experimental Set-up

One hundred and twenty 2-goal problems were randomly generated for Transportation domain and Softbot domain. For Transportation domain, each problem had two objects to deliver, three cities, three trucks and two planes. Softbot problems contained two persons about whom some information was sought. For the process planning domain, the number of goals for each of the 120 problems randomly ranged between 2 and 5. The process planning domain had two objects and the goal was to shape them.

Training sets of 20, 30, 40, and 60 were randomly selected from the 120-problem corpus, and for each training set, the remaining problems served as the corresponding testing set. To identify a model plan for each training problem, POP was run in a depth-first search mode with a depth limit of 15. The first 20 plans (or all possible solutions for a problem if this number was less than 20) were generated and the highest quality plan from these was used as a model plan for that problem. These were also the plans from which the

number of training examples		0	20	30	40	60
number of new nodes expanded	PIP-rewrite-first	24+0	9.8 + 6.7	8.3+ 6	7+ 7	7.8 + 5
	PIP-rewrite-best	24+0	9.8 + 36	8.9+ 44	8.5+ 75	7.9 + 95
	PIP	24	18.3	17.45	17.3	16.8
average difference from optimal quality plans	PIP-rewrite-first	1	0.82	0.84	0.78	0.80
	PIP-rewrite-best	1	0.01	0	0	0
	PIP	1	0.05	0.04	0.03	0

Table 4.1: Performance data for the process planning domain.

	num rules	num rules used	num rules revised
PIP-rewrite-first	3	2	n/a
PIP-rewrite-best	3	2.5	n/a
PIP	4	2	0

Table 4.2: Rule data for the process planning domain in the 20-problem case.

distance was measured to compute the plan quality metric. Planning effort was measured by the number of new nodes expanded by each planner. Rewrite module of PIP-rewrite-first uses the first-improvement search strategy and the rewrite module of PIP-rewrite-best uses the best-improvement search strategy as described in Chapter 3.

4.2.4 Results

Tables 4.1, 4.3 and 4.5 show the mean plan quality metric (i.e, m_Q as described in Equation 4.2) and the mean number of nodes expanded (i.e., m_{NumPP} as described in Equation 4.3) by PIP-rewrite and PIP on Softbot, process-planning and transportation domains, respectively. The new nodes expanded by PIP-rewrite are shown as $N + M$, where N is the mean number of nodes expanded by the default planner and M is the mean number of nodes expanded by the rewrite-module (i.e., the number of nodes required to refine the flaws introduced by applying rewrite rules to the initial plan). The two counts are represented separately because the rewrite nodes are slightly less costly than the planning nodes. This is because the rewrite module (shown in Figure 4.13) is a more restricted version of the partial-order planning module as it cannot add any new actions.

number of training examples		0	20	30	40	60
number of new nodes expanded	PIP-rewrite-first	36+0	14+132	14+156	13+124	12+127
	PIP-rewrite-best	36+0	14+14212	14+21518	13+22020	12+22788
	PIP	36	12.5	13	12	11
average difference from optimal quality plans	PIP-rewrite-first	1	0.95	0.96	0.94	0.92
	PIP-rewrite-best	1	0.85	0.74	0.72	0.70
	PIP	1	0.03	0.02	0.01	0

Table 4.3: Performance data for the transportation domain.

	num rules	num rules used	num rules revised
PIP-rewrite-first	12	4	n/a
PIP-rewrite-best	12	6	n/a
PIP	13	6	3

Table 4.4: Rule data for the transportation domain in the 20-problem case.

Tables 4.2, 4.4 and 4.6 display the mean number of rules learned by each system (m_{NumPP} as specified in Equation 4.3), the mean number of the rules that were used to construct a plan ($m_{NumUseful}$), and the mean number of the rules that lead to a lower quality plan and force PIP to learn a more specific rule ($m_{NumRevised}$). Each of these metrics was computed for the 20-problem sets; the problem sets that had the largest number of cross-validation runs. As specified in Section 4.2.1, the mean of each of these metrics was computed by measuring six values of each metric in the six cross-validation runs, computing a sum of these six values, and dividing it by six.

For all three domains, both rewrite and the search-control rules lead to substantial improvements in plan quality (i.e., reduction in the distance from

number of training examples		0	20	30	40	60
number of new nodes expanded	PIP-rewrite-first	10.4+0	3.4 + 21	3.0+ 22	2.5+25	2.1 + 24
	PIP-rewrite-best	10.4+0	3.4 + 86	3.0 +96	2.5+108	2.1 + 126
	PIP	10.4	3.03	3.0	2.44	2.1
average difference from optimal quality plans	PIP-rewrite-first	1	0.67	0.65	0.59	0.60
	PIP-rewrite-best	1	0.22	0.18	0.14	0.13
	PIP	1	0.55	0.47	0.14	0.12

Table 4.5: Performance data for the softbot domain.

	num rules	num rules used	num rules revised
PIP-rewrite-first	24	7	n/a
PIP-rewrite-best	24	11	n/a
PIP	24	15	9

Table 4.6: Rule data for the softbot domain in the 20-problem case.

the model plans as shown in Tables 4.1, 4.3, and 4.5). As expected, the quality of the plans produced by PIP-rewrite-best is higher than those produced by PIP-rewrite-first. It is interesting to note, however, that for all three domains, quality improvements obtained by using search-control rules are comparable or better than those obtained by rewrite rules (even when the entire neighborhood is exhaustively explored). For Softbot and the process planning domains, PIP-rewrite-best performs slightly better than PIP, whereas for the transportation domain the quality of PIP's plans is better than those produced by PIP-rewrite-best.

On the planning efficiency front, PIP clearly outperforms PIP-rewrite-best on all three domains. More surprisingly, PIP's performance on planning efficiency is even better than that of PIP-rewrite-first on two out of three domain. On the simple process planning domain, PIP-rewrite-first is more efficient than PIP but on the more interesting Transportation and Softbot domains PIP clearly outperforms PIP-rewrite-first.

4.2.5 Discussion

The empirical results presented here suggest that learning good rewrite rules using the PIP framework of analyzing local planning decisions is hard because it is difficult to translate information learned from one context (i.e., the context of choosing between plan refinement paths) into a form usable in another context (i.e., replacing portions of completed plans). I will illustrate this point with the help of the two Transportation examples introduced earlier (Problem 1 shown in Figure 3.3 and Problem 2 presented in Figure 3.19 and reproduced in Figure 4.9). Recall that on being trained on Problem 1, PIP learns Search Control Rule 1 and Search Control Rule 2 (shown in Figure

4.1) and PIP-rewrite learns the Rewrite Rule 1 (shown in Figure 4.2). Also recall that when subsequently Problem 2 is presented to PIP, it uses Search Control Rule 1 to produce the optimal quality plan {`load-plane(o1, pl1, ap1), fly-plane(pl1, ap1, ap2), unload-plane(o1, pl1, ap2)`}. PIP-rewrite, on the other hand, uses its Rewrite Rule 1 to produce the sub-optimal plan {`drive-truck(tr1, po1, ap1), load-plane(o1, pl1, ap1), fly-plane(pl1, ap1, ap2), unload-plane(o1, pl1, ap2)`} for the same problem.

The reason why PIP-rewrite fails to produce the optimal quality plan for Problem 2, despite having learned essentially the same information as the search control rule system, is that it applies this information *after* the complete plan has been produced. At that point, PIP-rewrite's default planner has traversed the suboptimal planning path to the end and may have added some more suboptimal actions during that process. For instance, in this case the base planner adds the extra action `drive-truck(tr1, po1, ap1)` which is not mentioned for deletion in the rewrite rule. And the reason it is not mentioned for deletion in the rule learned from Problem 1 is that in that problem, the action `drive-truck(r1, ap1, po1)` is not one of the relevant actions. The reason why search control rule learned from Problem 1 works in Problem 2 is precisely because it is applied *earlier during planning* to prevent the planner from going down the suboptimal planning path.

The above discussion seems to suggest that the search control rules learned by PIP are always more general than rewrite rules (learned from the same learning opportunities) because they apply early in the planning process. However, that is not always true[2]. To understand this, consider the problem shown in Figure 4.14 drawn from the softbot domain. The ISL algorithm identifies the conflicting choice point shown in Figure 4.14(b) when given the training problem shown in Figure 4.14(a). PIP turns the output returned by ISL into Search Control Rule 3 shown in Figure 4.14(d) while PIP-rewrite forms Rewrite Rule 3 shown in Figure 4.14(c).

---

[2] However, modifying the search control rule mechanism specified in Chapter 3 can change that, as I discuss shortly.

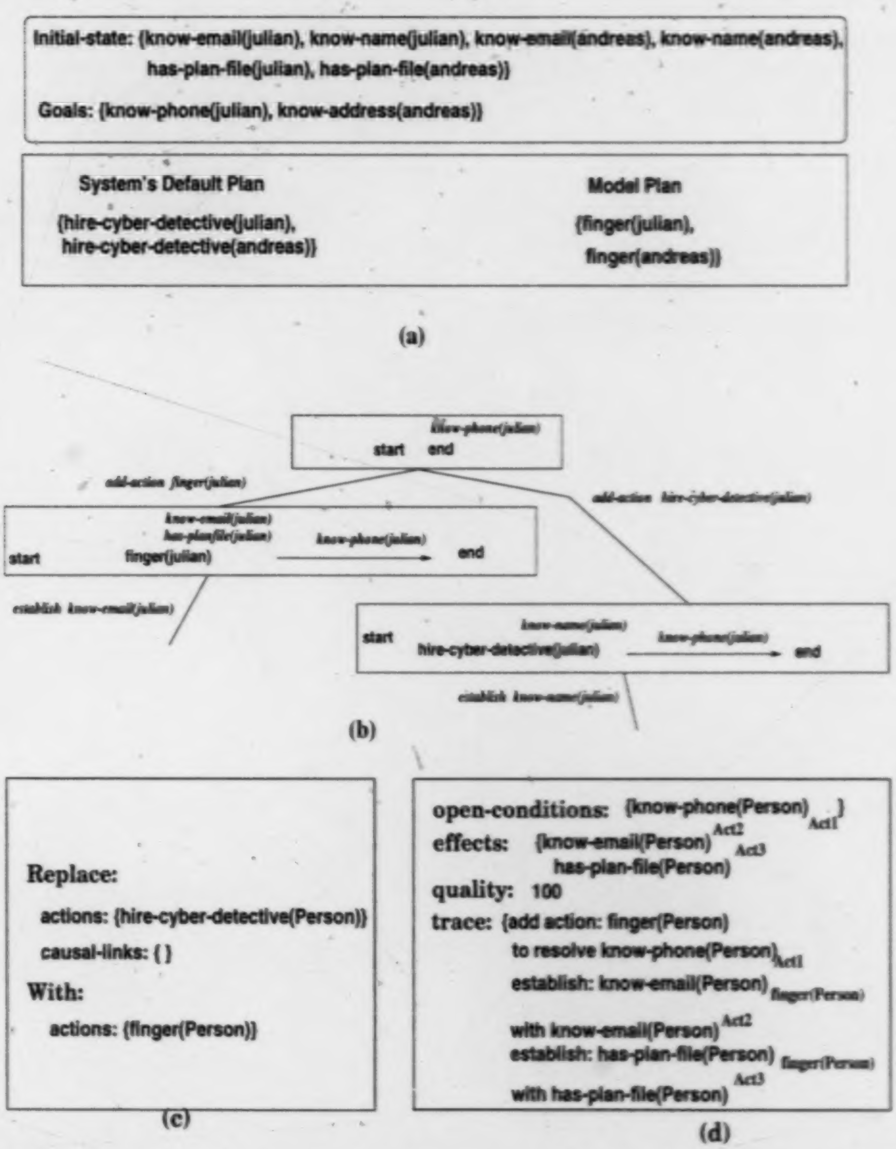
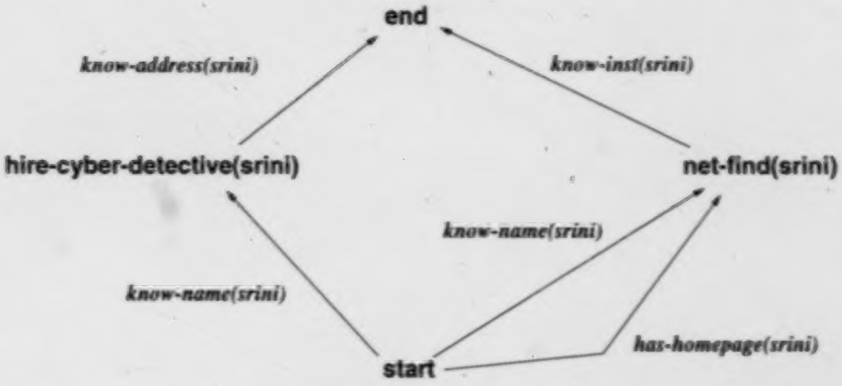

Initial-state: {know-email(julian), know-name(julian), know-email(andreas), know-name(andreas),
has-plan-file(julian), has-plan-file(andreas)}
Goals: {know-phone(julian), know-address(andreas)}
System's Default Plan
{hire-cyber-detective(julian),
hire-cyber-detective(andreas)}
Model Plan
{finger(julian),
finger(andreas)}
(a)
know-phone(julian)
start
end
add-action finger(julian)
add-action hire-cyber-detective(julian)
know-email(julian)
has-planfile(julian)
know-phone(julian)
start
finger(julian)
end
establish know-email(julian)
know-name(julian)
know-phone(julian)
start
hire-cyber-detective(julian)
end
establish know-name(julian)
(b)
Replace:
actions: {hire-cyber-detective(Person)}
causal-links: { }
With:
actions: {finger(Person)}
(c)
open-conditions: {know-phone(Person)_Act1}
effects: {know-email(Person)^Act2
has-plan-file(Person)^Act3
quality: 100
trace: {add action: finger(Person)
to resolve know-phone(Person)_Act1
establish: know-email(Person)_finger(Person)
with know-email(Person)^Act2
establish: has-plan-file(Person)_finger(Person)
with has-plan-file(Person)^Act3
(d)

Figure 4.14: (a) Problem 3: A training problem drawn from the softbot domain. (b) The conflicting choice point identified by ISL for the training problem shown in part (a). (c) Rewrite Rule 3: the rewrite rule learned by PIP-rewrite from the learning opportunity shown in part (b). (d) Search Control Rule 3: one of the two search control rules learned by PIP from the learning opportunity shown in part (b). This one specifies the rationale for using the better planning decision sequence. Pre_{Act} denotes precondition Pre of Action Act and Eff^{Act} denotes effect Eff supplied by the action Act.

Initial-state: {know-name(srini), has-email(srini), has-plan-file(srini), has-homepage(srini)}

Goals: {know-address(srini), know-inst(srini)}

(a)


end
know-address(srini)
know-inst(srini)
hire-cyber-detective(srini)
net-find(srini)
know-name(srini)
know-name(srini)
has-homepage(srini)
start

(b)

Figure 4.15: (a) Problem 4: A problem drawn from the softbot domain. (b) Planning graph corresponding to the plan {`hire-cyber-detective(srini)`, `netfind(srini)`} produced by PIP-rewrite's default planner for the problem shown in part (a).

Subsequently when Problem 4 (shown in Figure 4.15(a)) is presented to PIP, it cannot use the rationale that it learned from Problem 3 because that rationale can only be retrieved when *know-phone(Person)* is an open condition, and this never happens when solving the current problem. Rewrite Rule 3 however is applicable to the complete plan produced by PIP-rewrite's default planner (shown by the planning graph of Figure 4.15(b). This allows PIP-rewrite to produce the better quality plan {`finger(srini)`,`netfind(srini)`} for Problem 4. In this case, a rewrite rule is more general than the search control rule learned from the same learning opportunity because a search control rule is only indexed and retrieved by the goals it resolves in the example from which it was learned. This may account for PIP's poor performance in the softbot domain (especially when the number of training examples is small, as

open-conditions: {know-phone(Person)$_{Act1}$ or know-name(Person)$_{Act1}$
or know-address(Person)$_{Act1}$ }

effects: {know-email(Person)^{Act2}, has-plan-file(Person)^{Act3}

quality: 100

trace: {add action: finger(Person) to
to resolve know-phone(Person)$_{Act1}$
establish: know-email(Person)$_{finger(Person)}$
with know-email(Person)^{Act2}
establish: has-plan-file(Person)$_{finger(Person)}$
with has-plan-file(Person)^{Act3}

Figure 4.16: Modified form of Search Control Rule 3. Pre_{Act} denotes precondition Pre of Action Act and Eff^{Act} denotes effect Eff supplied by the action Act.

shown in Table 4.5).

This type of situation only occurs when the set of *available effects* of the planning decision sequence is larger than the goals the planning decision sequence was used to resolve in the example from which it was learned. This means that this planning decision sequence can also be used to resolve some other goals than the ones that PIP indexes it by. Available effects of a planning decision sequence are the effects supplied by an action added by the planning decision sequence that can be used to resolve the preconditions of an action that is not added as a part of the planning decision sequence. For example, the available effect set of the planning decision sequence stored in the `trace` part of the search control rule shown in Figure 4.14(d) is {`know-name(Person)`, `know-address(Person)`, `know-phone(Person)`, `know-inst(Person)`} which is larger than the open condition set by which PIP indexes this rule (namely {`know-phone(Person)`} as shown in Figure 4.14(d)).

Modifying PIP to index its search control rules by the disjunction of all the goals it can resolve can solve this problem. For instance, Search Control Rule 3 (shown in Figure 4.14(d)) would be transformed into the rule shown in Figure 4.16. This change should also ensure that search control rules are always more general than the rewrite rules learned from PIP's learning. I outline how this change is expected to affect PIP in Section 6.2.1. One of the expected benefits is that PIP should learn faster in domains such as Softbot and its rule memory should also become smaller.

4.3 Summary

This chapter presents PIP-rewrite, a variation of the PIP system presented in Chapter 3, that stores the information returned by PIP's learning component as rewrite rules (instead of search control rules). PIP-rewrite uses ISL to identify two subplans, a bad subplan which can be replaced by the other good subplan. After learning this information, whenever PIP-rewrite produces an initial plan containing the bad subplan it tries to replace it with the good subplan to produce a plan which is hopefully of a better quality than the initial plan. This planning by rewriting framework has the potential of improving the planning efficiency as well, because the initial plans are produced by a speed-up planner called DerPOP. Experimental evidence is presented in this chapter to show that learned rewrite rules do lead to improvements in plan quality on a number of benchmark planning domains. However, the gains in efficiency made by using a speed-up planner to generate the initial plan are lost during the rewrite process. The empirical results also show that there is information to be gained by analyzing local refinement decisions during the planning process and translating them into both rewrite as well as search control rules. However, it appears that rewrite rules, by their very nature of working on completed plans, just do not have the same decision-making context.

There are also significant differences in PIP's performance across different benchmark domains. This suggests that there may be some domain features varying which can affect PIP's performance improvements. To understand how various domain features affect PIP's performance, I designed various artificial planning domains [BW94] in which I could systematically vary various features and understand how they affect PIP's performance. These experiments and their results along with PIP's comparison with other systems that improve plan quality for partial order planners are presented in the next chapter.

Chapter 5

Evaluating PIP

Comparisons with competitor systems to evaluate the performance of a system are a standard part of the evaluation of AI systems. Besides PIP, SCOPE [Est98] is the only planning and learning system that automatically learns to improve quality of the plans produced by partial order planners. The first part of this chapter presents results of the empirical comparison of PIP and SCOPE.

The rest of this chapter analyzes PIP's learning module, ISL. I argue that viewing ISL as a supervised concept learner gives us some guidance on how to evaluate PIP. A number of domain features are identified that are likely to have an impact on PIP's performance. I describe the experimental set up and provide results of the experiments done to evaluate the impact of varying domain features on PIP's performance.

5.1 Empirical Comparison With SCOPE

SCOPE [EM97, Est98] is the only planning and learning system besides PIP that learns to improve quality of the plans produced by partial-order planners. As described in Sections 2.2.1 and 2.3.3, SCOPE uses inductive learning techniques to acquire search control rules to improve plan quality. SCOPE does not possess plan quality knowledge, hence it can only learn quality improving rules in the apprenticeship learning mode (i.e., when the better quality model plans are provided by a domain expert). PIP, on the other hand, can generate alternative plans automatically, evaluate their quality, and learn if their qualities are different.

The conventional wisdom in machine learning is that when all else is equal, analytic techniques require fewer training examples than inductive techniques. However, as described earlier (Section 2.2.1), SCOPE uses the search-tree to limit the language for the concepts it learns. SCOPE also uses some domain specific concepts such as the concept of ***above(Block1, Block2)*** defined for Blocksworld problems. This makes a theoretical comparison of the two systems very difficult. [EM97] presents experimental results to show that SCOPE can improve quality (defined as plan length) of the plans produced by the partial-order planner UCPOP. I repeated those experiments for PIP. The improvements in plan quality and planning efficiency obtained by PIP were then compared with those reported for SCOPE in [EM97].

[EM97] used average plan length as the plan quality metric while planning efficiency was measured by computing the CPU time. Since plan length was the plan quality criterion, both PIP and SCOPE used depth-first search as the search strategy[1]. Depth-first iterative deepening (DFID) was used to produce the model quality solutions for the training problems.

5.1.1 Experimental Set-up

Veloso's logistics transportation domain was used for these experiments. Five problem sets of size 100 were generated. Each problem contained one or two objects to deliver, two trucks, and two planes which were distributed among two cities. Both PIP and SCOPE were trained on example sets of increasing size (10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 problems) for all the five problem sets and the results were averaged.

5.1.2 Results

The platform used to run PIP for these experiments was a Sun/Sparc Ultra 1 machine. Figure 5.1 shows the average planning time (not counting the rule

---

[1] This done because using Depth-First Iterative Deepening (DFID) [Kor85] as default search strategy would have meant that the systems could produce the optimal plans without any learning.

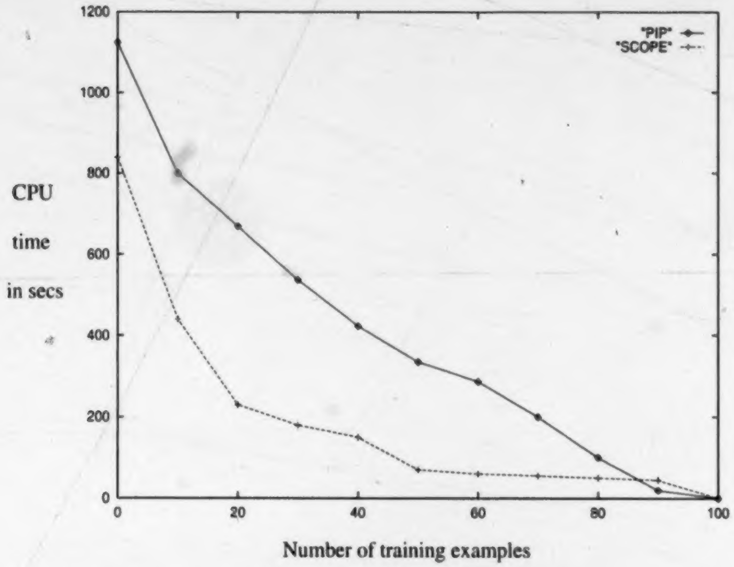


Figure 5.1: Graph showing how PIP and SCOPE improve planning efficiency. The graph for SCOPE is reproduced from [EM97].

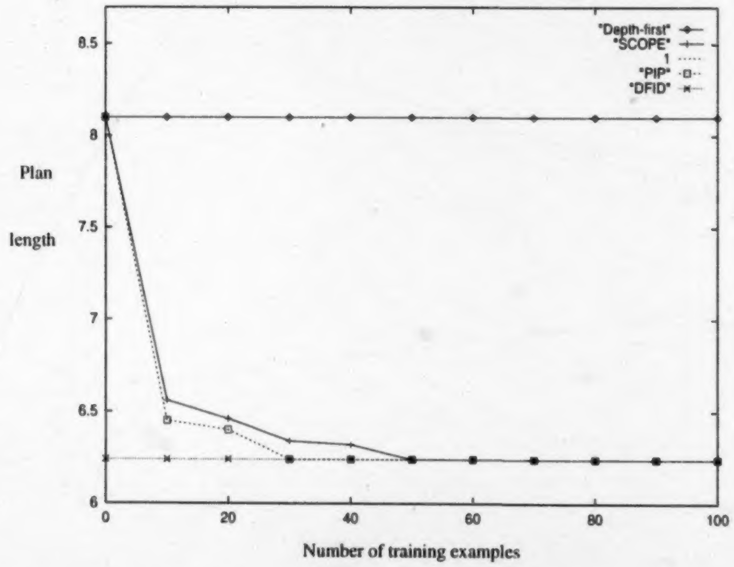


Figure 5.2: Graph showing how PIP and SCOPE improve plan quality as the number of training examples increases. The graph for SCOPE is reproduced from [EM97].

retrieval time) required by PIP and the average planning time required by SCOPE to produce a plan for the test problems. As the number of training examples increases the amount of time taken by both systems decreases. However, it is difficult to draw any conclusions by comparing the planning times of two algorithms because of the differences in the compilers, platforms, and implementation techniques. Clearly, SCOPE's base planner UCPOP is more efficient than PIP's base planner POP because SCOPE's planning time with no learning is significantly better than PIP's time with no learning. But that does not tell us much about the relative performance of the two *learning algorithms* which is what we are interested in.

Figure 5.2 shows the average plan length of the plans produced by each planning system. The average length of the plan produced by depth-first search remains unchanged at 8.1 and shows the baseline performance without learning. The line corresponding to DFID shows the optimal performance. The results show that PIP needs only 30 examples to reach the optimal performance while SCOPE needs almost twice as many (50) examples to converge to the optimal performance. This confirms our intuitions that analytic techniques require fewer examples to learn a concept than inductive techniques.

Note that this empirical comparison does not take advantage of PIP's enhanced representation of plan quality. The reason being that SCOPE is not designed to handle quality as a function of multiple factors. PIP, on the other hand, is designed to improve its performance on complex plan quality measures involving a number of variables. PIP's performance on quality is not affected by the number of variables involved in the plan quality function. It is affected by a number of other factors such as *instance similarity* and *quality branching factor*. The next section discusses various factors that affect PIP's performance, and reports on empirical experiments done to evaluate how these factors affect PIP's performance on plan quality.

5.2 Analysis of Factors That Affect PIP's Performance

5.2.1 PIP's Learning Component, ISL, As A Supervised Concept Learner

As discussed in 4.2.5, the learning space of PIP's learning component (ISL) consists of conflicting choice points. Each conflicting choice point is composed of a partial plan, with an associated good planning decision and an associated bad planning decision. The target concept that ISL must learn from these examples is which planning decision to apply to a partial plan to resolve its flaws. The approximation of this target concept learned by ISL (encoded into its if-then rules) is then used to classify the partial plans generated during its search for solutions for the test examples to produce better solutions (according to the performance measure defined on the learning system's task) than the solutions the system would have produced without any learning. The set of all partial plans that can possibly be generated by the planning problems in that domain defines the instance space of ISL. Viewing ISL as a supervised concept learner as described above allows us to better understand it. It also gives us some guidance on how to evaluate its performance.

5.2.2 Factors For Evaluating Supervised Learning Algorithms

Computational learning theory tells us that a supervised concept learner can only be guaranteed to perform well on some unseen test problems if the distribution of the test examples is identical to that of the training examples (the so-called *stationary assumption*) [KV94]. This is intuitive because in the trivial case, if the test problems are completely unrelated to the training problems, then the learned knowledge cannot be of any use in solving the test problems. On the other hand, if the learner is presented with the same problems in the training and testing phase then the learner can be expected to perform very well. In general, the greater the similarity between the training and the test examples, the more useful the learned knowledge can be and the better a learner can be expected to perform. Instance similarity was the first factor I decided to vary to assess the impact of similarity on PIP's performance.

The second factor varied in the following experiments was the *quality branching factor*. Quality branching factor is defined as the average number of different quality plans per problem in the domain of interest. In the trivial case when the quality branching factor is zero (i.e., the quality of all the solutions is alike), it is easy to see that PIP's learning mechanism will never be invoked hence it will not learn anything.

The third factor considered is the association between the planner's *default bias* and the *quality bias*. Quality bias is the knowledge about the target concept that PIP is trying to learn. In order to efficiently generate plans, domain independent planners often assume some domain independent biases. I call these the planner's *default biases*. For instance, UCPOP has a default bias to explore those partial plans first that have a lower value of $A + C + T$, where A is the number of actions, C is the number of open-conditions and T is the number of threats present in the partial plan. If the planner's default biases are close to the target quality bias (i.e., the system is lucky) then there is not much to learn because the system can produce good quality solutions without any learning.

5.2.3 Empirical Experiments Using Artificial Domains

Problem set and domain generation

Artificial domains and problem sets were generated to evaluate how varying each of the factors discussed in the last section affects PIP's performance. The problem set generation algorithm had two parameters: the size of the *possible goal set* and the size of the *possible initial condition-set*. The number of initial conditions for each problem was set to five and the number of goals for each problem was set to three. The possible goal set consists of all the possible goal propositions from which the problem generator has to select three goals for all the problems to be generated. The possible initial condition set contains all the propositions from which the problem generator randomly selects five initial condition propositions for each problem.

The domain generation algorithm has two parameters: the *possible precondition set* and the *possible effect set*. The possible precondition set contains all the preconditions from which preconditions for the domain actions are randomly selected and the possible effect set contains all the effects from which effects for the domain actions are randomly selected. The possible precondition set is a superset of the possible initial condition set, and the possible effect set contains all the possible goals. The total number of actions was set to 18. Actions a_1 to a_6 added the goals (which were randomly selected from the possible goal set), actions a_{13} to a_{18} required all the initial conditions as their preconditions, and actions a_7 to a_{12} added *intermediate* preconditions and effects. Intermediate preconditions are the preconditions that are present in the set of possible preconditions but not in the set of possible initial conditions. Intermediate effects are the effects that are present in the set of possible effects but absent from the set of possible goals. Appendix D shows the domain used for the experiments done by varying problem similarity. The possible precondition set for this domain, $\{i_1, i_2, \ldots, i_{12},\ p_1, p_2, \ldots, p_{12},\ q_1, q_2, \ldots, q_{12}\}$, contains all the possible initial conditions $i_x, x = 1, 2, \ldots, 12$ and the intermediate preconditions p_y and q_z where $y, z = 1, 2, \ldots, 12$. The possible effect set for this domain, $\{g_1, g_2, \ldots, g_{12},\ p_1, p_2, \ldots, p_{12},\ q_1, q_2, \ldots, q_{12}\}$, contains all the possible goals $g_x, x = 1, 2, \ldots, 12$ as well as all the intermediate effects p_y and q_z where $y, z = 1, 2, \ldots, 12$.

Methodology

A total of 120 unique problems were generated using the problem generation algorithm described earlier. The experimental methodology of cross-validation (described earlier in Section 4.2.1 and used for experiments with benchmark domains in the last chapter) was followed in the experiments reported here. Recall that cross-validation procedure for an x-item ($x = 20, 30, 45, 60$) training set is as follows: there are $\frac{120}{x}$ unique runs, each defined by a unique set of x-training items and $\frac{120}{x}$ testing items. This ensures that after all the $\frac{120}{x}$ runs, each of the total of 120 problems has appeared $\frac{120}{x}$ times as a test problem.

So there are $\frac{120}{20} = 6$ cross validation runs in case of the 20-problem set, $\frac{120}{30} = 4$ cross validation runs in case of the 30-problem sets, $\frac{120}{40} = 3$ cross validation runs for the 40-problem sets, and $\frac{120}{60} = 2$ cross validation runs in case of the 60-problem sets.

Metrics of interest. The normalized average distance from optimal quality plans, Q_3 (as specified in Section 4.2.1) was used to measure PIP's plan quality. As described in Section 4.2.1, the value of Q_3 was calculated for each of the $\frac{120}{x}$ cross validation runs for an x-problem set. These values were used to compute the mean plan quality metric (m_Q defined in Equation 4.2). Standard deviation among the six Q_3 values computed for each of the six cross-validation runs in the 20-problem case was also computed to provide a measure of the spread of the Q_3 values. The planning efficiency metric used for the following experiments was the mean number of new nodes PIP's planner expands to solve the testing problems (m_{NumPP} defined in Equation 4.3).

The metrics used for measuring the rule utility included *the proportion of useful rules* in the 20-problem case, and *the proportion of rules needing refinement* in the 20-problem case. The proportion of useful rules is defined as the number of rules that were used by PIP at least once for construction of a plan for a subsequent problem divided by the total number of rules learned by PIP during that run (i.e., $\frac{number\ of\ rules\ used\ at\ least\ once}{number\ of\ rules\ learned}$). The proportion of the rules needing revision is defined as the number of rules that need to be refined because they lead to a lower quality plan (than the model plan for that problem) during training, divided by the total number of rules that PIP learned for that run (i.e., $\frac{number\ of\ rules\ refined}{number\ of\ rules\ learned}$). The value of each of both these metrics (i.e., the proportion of useful rules, and the proportion of rules needing refinement) was calculated for each of the six cross validation runs for a 20-problem set. A mean and standard deviation of these six values was then computed for each of these metrics.

5.2.4 Varying Instance Similarity

ISL's instance space for a problem set (i.e., the partial plans generated during the search for solutions for the problems in that set) is completely determined by the problem descriptions as well as by the domain descriptions. This is because the problem specifications (i.e., the initial-state and the goals) only completely determine the *initial* partial plans and not the *intermediate* partial plans which are also partly determined by the preconditions and the effects of the domain actions. To test the effect of varying instance similarity on PIP's performance, the following two factors were varied:

- problem description (i.e., initial-condition and goal) similarity
- precondition and effect similarity

The greater the amount of similarity of the problems within a domain, the greater the chance that similar partial plans will be generated during the search.

Increasing the domain similarity, defined this way, has the desirable effect of increasing the similarity between training items and the testing items. This means that more knowledge learned during the training phase will be applicable during the testing phase which should improve PIP's performance. However, it also has the unintended effect of making all items (i.e., the training items as well as the testing items) *internally* similar (i.e., one training item similar to another training item and one testing item similar to another testing item). When training items belonging to different concepts are similar to one another, it is harder for a concept learner to learn their distinguishing features. When testing items belonging to different classes are similar to one another, there are greater chances of misclassification (i.e., an item belonging to class A being placed in class B).

In PIP's case, a misclassification means retrieval and application of a rule to a partial plan that leads it to the production of a lower quality plan. A rule is applicable to a partial plan that contains the open conditions and effects required by the planning decisions stored in the rule. As described in

Section 3.2.4, of all the applicable rules, PIP retrieves the rule that promises the highest quality way of resolving the largest number of its open-condition flaws. A retrieved rule may provide wrong guidance (i.e., lead to a lower quality plan) if the partial plan that retrieved the rule contains some open conditions that negatively interact with the rule's open conditions. Unfortunately, as the internal similarity between training/testing items increases, the likelihood of two partial plans being generated that have some (but not all) of their open conditions in common also increases. The larger the number of partial plans that have some (but not all) of their open conditions in common with other partial plans, the greater the number of negatively interacting partial plans. The greater the likelihood of the generation of negatively interacting partial plans, the larger the number of rules that provide wrong guidance and need to be refined. In short:

- learning is not likely to be very useful for solving subsequent problems when partial plans generated are very dissimilar.
- learned knowledge is likely to be more useful in solving subsequent problem when the instances are similar (i.e., more percentage of rules will be used). On the other hand, finer discriminations between the partial plans must be made to decide which planning decisions to apply i.e., which rule to retrieve and apply. Thus an increase in instance similarity may also increase the chances of wrong rules being applied.

Given this discussion, I propose the following three testable hypotheses.

Hypothesis 1 *More of PIP's knowledge will be useful as the instance similarity increases.*

Hypothesis 2 *More of PIP's knowledge will need to be refined as the instance similarity increases.*

Hypothesis 3 *The amount of improvement in PIP's plan quality will initially increase as the instance similarity increases (and more search control rules are applied), then drop as the instance similarity further increases (and more rules are wrongly applied).*

Normalized average distance of PIP's plans from the optimal quality plans is the measure used to determine plan quality. The value of this metric drops as PIP's plan quality increases and it increases as PIP's plan quality drops. The *usefulness* of PIP's knowledge is determined by measuring the proportion of useful rules. The amount of PIP's knowledge that needs to be revised is determined by measuring the proportion of the rules that need to be refined.

Varying Problem Description Similarity

A planner such as POP considers two problems to be similar if their initial conditions and goals are similar. The chances of two problems with similar initial conditions and goals being generated by the problem generation algorithm depend on the total number of unique problem descriptions, i.e., size of the set of possible initial conditions and the size of the set of possible goals from which the 120 unique problems are to be randomly selected.

Experimental Set up. The problem set generation algorithm described in Section 5.2.3 was used to generate nine problem sets to test Hypotheses 1-3. The problem similarity was varied by varying both the number of possible initial conditions as well as the number of possible goals from 6 to 12. The most similar problem set had 120 unique problems (with twenty unique 3-goal sets, and six initial condition sets of size 5) while the most different problem set had 174240 unique problems (with 220 unique 3-goal sets and 792 5-goal problem sets) from which 120 problems could be selected randomly.

The domain set was generated using the domain generation algorithm described in Section 5.2.3 with size of the possible precondition set being 12 and the possible effect set size being 12. The domain generation algorithm was repeatedly invoked until a domain was generated that allowed PIP to solve all the 120 problems in the most similar problem set. The domain was then fixed and used for experiments with all other problem sets.

Results. The plan quality data shown in Table 5.1 is the mean value of the plan quality metric m_q (specified in Equation 4.2). The planning efficiency

num init-conds	num goals	Number of training examples				
		0	20	30	40	60
6	6	1	0.78	0.75	0.71	0.65
6	9	1	0.71	0.68	0.47	0.38
6	12	1	0.69	0.64	0.55	0.56
9	6	1	0.77	0.70	0.64	0.49
9	9	1	0.61	0.39	0.23	0.18
9	12	1	0.80	0.82	0.77	0.70
12	6	1	0.68	0.66	0.55	0.22
12	9	1	0.66	0.67	0.59	0.30
12	12	1	0.83	0.74	0.80	0.75

Table 5.1: Mean plan quality metric as a function of problem similarity and training set size. The table shows how the normalized distance from the optimal quality plans changes as the number of training problems is increased from 0 to 60 for all nine problem similarity domains.

data shown in Table 5.2 is the mean number of new partial plans m_{NumPP} (specified in Equation 4.3). The rest of the tables (Tables 5.3–5.5) present more data for the 20-problem sets: the problem sets with the largest number of cross-validation runs. Recall that in the 20-problem case, the 120 problem set is divided into 6 unique sets, each having 20 training problems and 100 testing problems. PIP is then run on each of these sets and the performance metrics measured for each run. This leaves us with six values of each of the performance metrics (namely, Q_3, the average number of new partial plans generated per problem, the proportion of useful rules, and the proportion of these rules that need refinement). Mean and standard deviation of the six values of each metric, measured from the six cross validation runs, were then calculated. Table 5.3 shows the mean and the standard deviation for the proportion of the useful rules. Table 5.4 tabulates the mean and the standard deviation for the proportion of the rules needing refinement in the 20-problem case. Finally, Table 5.5 shows the mean and standard deviation for the plan quality metric.

Six one-tailed t-tests were performed to test each of the three hypotheses presented in the last section (namely, Hypothesis 1, Hypothesis 2, and Hypothesis 3). The first two of these tests are intended to study the effect of

num init-conds	num goals	Number of training examples				
		0	20	30	40	60
6	6	24.8	12	10.4	10.5	9.7
6	9	28.6	17.8	19.7	18.1	16.9
6	12	29	18.1	19	19.5	17
9	6	28.3	18.3	19.2	20.2	16.9
9	9	32.5	22	23.7	24	20.5
9	12	35.1	26.9	27.6	28	26.4
12	6	31.6	20.4	18	18.8	16.3
12	9	28.7	18.5	20.9	19.6	17
12	12	38.3	28.5	29.5	29.9	24.9

Table 5.2: Mean planning efficiency metric as a function of problem similarity and training set size. The table shows how the average number of new search nodes changes as the number of training problems is increased from 0 to 60 for all nine problem similarity domains.

			high similarity		low similarity
			Number of possible goals		
			6	9	12
high similarity	Number of	6	1.00 (0.10)	0.71 (0.13)	0.65 (0.10)
	possible	9	0.65 (0.11)	0.88 (0.12)	0.48 (0.08)
low similarity	init-conds	12	0.68 (0.10)	0.50 (0.08)	0.28 (0.07)

Table 5.3: Mean and standard deviation (in parenthesis) of the proportion of the useful rules in the 20-problem case as a function of problem similarity.

			high similarity		low similarity
			Number of possible goals		
			6	9	12
high similarity	Number of	6	0.15 (0.05)	0.10 (0.04)	0.05 (0.02)
	possible	9	0.08 (0.04)	0 (0.00)	0 (0.00)
low similarity	init-conds	12	0.04 (0.02)	0 (0.00)	0 (0.00)

Table 5.4: Mean and standard deviation (in parenthesis) of the proportion of the rules needing refinement in the 20-problem case as a function of problem similarity.

			high similarity		low similarity
			Number of possible goals		
			6	9	12
high similarity	Number of possible init-conds	6	0.78 (0.036)	0.71 (0.039)	0.69 (0.105)
		9	0.77 (0.095)	0.61 (0.092)	0.80 (0.043)
low similarity		12	0.68 (0.050)	0.66 (0.044)	0.83 (0.040)

Table 5.5: Mean and standard deviation (in parenthesis) of the plan quality metric in the 20-problem case as a function of problem similarity.

varying goal similarity on PIP's performance while the next two tests assess the impact of varying initial condition similarity. A third set of two t-tests was performed to assess what impact varying both these factors together has on PIP's performance.

The data in Table 5.3 formed the basis of the significance tests performed to test Hypothesis I regarding the mean proportion of the useful rules. The first set of t-tests compared means going across the first row in Table 5.3 to evaluate the impact of decreasing goal similarity on the proportion of useful rules. It compared the mean proportion of the rules needing refinement in the 6-6 case with the mean proportion of the rules needing refinement in the 6-9 case, and the mean proportion of the rules needing refinement in the 6-9 case with the mean proportion of the rules needing refinement in the 6-12 case. It was found that the mean proportion of rules that prove useful for subsequent planning in the 6-6 set (1.00) is significantly greater than the mean proportion of rules that are useful for subsequent planning in the 6-9 case (0.71) [$t = 4.33,\ p < 0.05$], and the 6-9 mean (0.71) is larger than the mean proportion of the useful rules in the 6-12 case (0.65), although not significantly so [$t = 0.90$]. Thus varying goal similarity has some impact on the proportion of useful rules learned by PIP.

The second set of tests compared the means going down the leftmost column in Table 5.3 to evaluate the impact of decreasing initial condition similarity on the proportion of useful rules. It was found that the mean proportion of rules that are useful for subsequent planning in the 6-6 case (1.00) is significantly higher than the mean proportion of useful rules learned in 9-6 case

(0.65) [$t = 5.77$, $p < 0.05$]. However, the mean proportion of useful rules in the 9-6 case (0.65) is less than the mean proportion of useful rules in the 12-6 case (0.68). The difference between the two means was not found to be statistically significant [$t = 0.49$].

A third set of tests compared the means going down along the diagonal of Table 5.3 to assess how decreasing both the initial condition similarity and the goal similarity affects the mean proportion of the useful rules learned by PIP. A one tailed t-test found that the mean proportion of useful rules for the 6-6 case (1.00) is significantly larger than the mean proportion of the useful rules for the 9-9 case (0.88) [$t = 1.88$, $p < 0.05$]. Another t-test found that the mean proportion of rules that are useful for subsequent planning in the more similar 9-9 case (0.88) is significantly greater than the mean proportion of rules that are useful for subsequent planning in the less similar 12-12 case (0.28) [$t = 10.58$, $p < 0.05$]. Thus decreasing both initial condition and goal similarity together significantly decreases the mean proportion of useful rules. This is what was predicted by Hypothesis 1.

The data in Table 5.4 formed the basis of the significance tests performed to test Hypothesis 2 regarding the mean proportion of rules that lead to lower quality plans and hence need to be refined. The first set of t-tests compared means going across the first row in Table 5.4 to evaluate the impact of decreasing goal similarity on the proportion of rules needing refinement. The mean proportion of the rules needing refinement in the 6-6 case (0.15) was found to be significantly greater than the mean proportion of the rules needing refinement in the 6-9 case (0.10) [$t = 1.91$, $p < 0.05$]. Similarly, the mean proportion of rules needing refinement in the 6-9 case (0.10) was significantly greater than the mean proportion of the rules needing refinement in the 6-12 case (0.05) [$t = 2.74$, $p < 0.05$]. From these two t-tests, we can see that decreasing goal similarity decreases the proportion of rules needing refinement.

The second set of t-tests compared the means going down the leftmost column in Table 5.4 to evaluate the impact of decreasing initial condition similarity on the proportion of rules needing refinement. It found that the mean proportion of the rules needing refinement in the 6-6 case (0.15) is significantly larger than the mean proportion of the rules needing refinement in the 9-6 case (0.08) [$t = 2.68$, $p < 0.05$], and that the mean proportion of the rules needing refinement in the 9-6 case (0.08) is significantly larger than the mean proportion of rules needing refinement in the 12-6 case (0.04) [$t = 2.19$, $p < 0.05$]. Thus decreasing initial condition similarity decreases the proportion of rules needing refinement.

A third set of t-tests compared the means going down along the diagonal of Table 5.4 to assess the impact of increasing both the initial condition and the goal similarity on the mean proportion of rules that PIP learns that need to be refined. A t-test found that the mean proportion of rules needing refinement in the 6-6 case (0.15) is significantly larger than the mean proportion of the rules needing refinement in the 9-9 case (0) [$t = 7.35$, $p < 0.05$]. No rules need refinement in the 9-9 case. The mean proportion of rules needing refinement cannot possibly decrease any further (i.e., the number of rules needing refinement cannot drop below zero) hence no further decrease in the mean proportion of rules was expected as the problem similarity is decreased to 12-12. This is what was observed. Thus decreasing both initial condition and goal similarity decreases the proportion of rules needing refinement as predicted by Hypothesis 2.

The data in Table 5.5 formed the basis of the significance tests performed to test Hypothesis 2 regarding improvements in plan quality obtained by PIP. The first set of tests compared the means going across the top row of Table 5.5 to evaluate how varying the goal similarity affects PIP's performance on plan quality. It was found that the mean value of the plan quality metric (i.e., normalized distance from the optimal quality plans) obtained in the 6-6 case (0.78) is significantly worse[2] than the mean value of the plan quality metric obtained for the less similar 6-9 set (0.71) [$t = 3.23$, $p < 0.05$]. The mean value of the normalized distance from optimal quality plans increases further as problem similarity is increased to 6-12. However, a t-test found that the difference between mean values of the plan quality metric in the 6-12 set (0.69) and the 6-9 set (0.71) is not statistically significant [$t = 0.44$].

---

[2] Since plan quality metric measures the normalized distance from the optimal quality plans, larger values of the plan quality metric are worse than the smaller values.

A second set of t-tests compared the means going down the leftmost column of Table 5.5 to evaluate the impact of decreasing problem similarity on PIP's performance with respect to plan quality. A t-test showed that the mean value of the plan quality metric in the 9-6 case (0.77) is not significantly different from the mean value of the plan quality metric obtained in the 6-6 set (0.78) [$t = 0.24$]. Another t-test showed that mean value of the plan quality metric (i.e., the normalized distance from the optimal quality plans) in the 12-6 set (0.68) is significantly better than the mean value of the plan quality metric obtained in the 9-6 case (0.77) [$t = 2.05,\ p < 0.05$]. Thus decreasing initial condition and goal similarity alone has some impact on PIP's performance with respect to plan quality.

A third set of tests compared the means going down along the diagonal of Table 5.5 to assess the impact of increasing both the initial condition and the goal similarity on the mean value of the plan quality metric. It was observed that the mean value of the plan quality metric (i.e., the normalized distance from optimal quality plans) in the more similar 6-6 case (0.78) is significantly worse than the mean value of the plan quality metric obtained in the less similar 9-9 case (0.61) [$t = 4.21,\ p < 0.05$]. Another t-test indicated that the mean value of the plan quality metric in the 9-9 case (0.61) is significantly better than the mean value of the plan quality metric in the 12-12 case (0.83) [$t = 5.37,\ p < 0.05$]. This means that as both initial condition and goal similarity are increased, PIP's performance with respect to plan quality improves initially and then drops as the problem similarity is further increased. This is what was predicted by Hypothesis 3.

Varying Precondition and Effect Similarity

Precondition/effect similarity of a domain is defined as the *average pairwise similarity* between the precondition/effect sets of two *competing* actions in the domain. Two actions are said to be competing if they have at least one precondition/effect in common. Similarity between two precondition/effect sets is defined as the percentage of the preconditions/effects the two actions share. The average pairwise similarity is computed by summing the similarity between all the competing actions and then dividing them by the number of such pairs.

Experimental Set Up. The domain generation algorithm described in Section 5.2.3 was used to generate the nine domains used to test Hypotheses 1-3 (about the effect of varying domain similarity on PIP's performance).

The size of the possible initial condition set and the size of the possible goal set was fixed at 6 and 9 respectively, to generate a problem set containing one hundred twenty problems (each consisting of 3 goals and 5 initial conditions). This problem set was then fixed and used for all nine domains. In order to ensure that each domain solved all 120 problems, the domain generation algorithm was repeatedly invoked until a domain was generated that allowed PIP to solve all of the 120 problems. The precondition and effect similarity for this domain was then measured and reported.

Results. The plan quality data reported here is the mean value of the plan quality metric m_q (specified in Equation 4.2) for all nine domains. Table 5.6 shows how the distance from the model plans changes as the number of training problems is increased from 0 to 60. The planning efficiency data reported in this section is the mean number of new partial plans m_{NumPP} (specified in Equation 4.3). Table 5.7 displays how the number of new search nodes generated changes as the number of training problems is increased from 0 to 60 for all nine domains. Rest of the tables present more data for the 20-problem sets: the problem sets with the largest number of cross-validation runs. Table 5.8 shows the mean and the standard deviation for the proportion of the rules used. Table 5.9 tabulates the mean and the standard deviation for the proportion of the rules that need to be refined in the 20-problem case. Finally, Table 5.10 shows the mean and the standard deviation among the values of the plan quality metric for the 20-problem sets.

Six one-tailed t-tests were performed to test each of the three hypotheses presented earlier (namely, Hypothesis 1, 2, and 3). The first two of these tests

precond similarity	effect similarity	Number of training examples				
		0	20	30	40	60
30%	45%	1	0.99	0.83	0.69	0.61
30%	70%	1	0.70	0.53	0.50	0.47
30%	80%	1	0.38	0.32	0.33	0.30
40%	45%	1	0.80	0.76	0.69	0.60
40%	70%	1	0.58	0.22	0.23	0.10
40%	80%	1	0.28	0.30	0.24	0.09
50%	45%	1	0.60	0.48	0.45	0.42
50%	70%	1	0.40	0.33	0.30	0.10
50 %	80%	1	0.42	0.38	0.45	0.37

Table 5.6: Mean plan quality metric as a function of domain similarity and the training set size. The table shows how the normalized distance from the optimal quality plans changes as the number of training problems is increased from 0 to 60 for all nine domains.

precond similarity	effect similarity	Number of training examples				
		0	20	30	40	60
30%	45%	67.5	54.4	55.6	46.2	26.3
30%	70%	25.2	21.2	19.6	18.3	17.1
30%	80%	16.1	17.2	13.1	12.7	12.5
40%	45%	55	48.6	33.5	31.1	26
40%	70%	29.6	14.9	12.9	14.8	14.9
40%	80%	15.6	13.3	10.6	10.1	10.2
50%	45%	43	35	24	22	20
50%	70%	22.4	15.6	18.4	17.9	15
50%	80%	14.5	9.5	7.6	7.3	6

Table 5.7: Mean planning efficiency metric as function of domain similarity and the training set size. The table shows how the mean number of new search nodes changes as the number of training problems is increased from 0 to 60 for all nine domains.

			Effect similarity		
			low similarity		high similarity
			45%	70%	80%
low similarity	Precond similarity	30%	0.40 (0.04)	0.45 (0.05)	0.60 (0.08)
		40%	0.50 (0.04)	0.50 (0.04)	0.63 (0.09)
high similarity		50%	0.63 (0.08)	0.61 (0.11)	0.65 (0.10)

Table 5.8: Mean and standard deviation (in parenthesis) of the proportion of useful rules in the 20-problem case as a function of domain similarity.

			Effect similarity		
			low similarity		high similarity
			45%	70%	80%
low similarity	Precond similarity	30%	0.05 (0.02)	0.10 (0.02)	0.15 (0.04)
		40%	0.10 (0.04)	0.10 (0.04)	0.15 (0.06)
high similarity		50%	0.15 (0.05)	0.17 (0.10)	0.20 (0.09)

Table 5.9: Mean and standard deviation (in parenthesis) of the proportion of rules needing refinement in the 20-problem case as a function of domain similarity.

			low similarity		high similarity
			Effect similarity		
			45%	70%	80%
low similarity		30%	0.99 (0.15)	0.70 (0.11)	0.38 (0.11)
	Precond	40%	0.80 (0.12)	0.58 (0.10)	0.28 (0.10)
high similarity	similarity	50%	0.60 (0.09)	0.40 (0.10)	0.42 (0.08)

Table 5.10: Mean and standard deviation (in parenthesis) of the plan quality metric in the 20-problem case as a function of domain similarity.

were intended to study the effect of increasing effect similarity on PIP's performance while the next two tests assessed the impact of increasing precondition similarity on PIP's performance with respect to plan quality. A third set of two t-tests was then performed to assess what impact increasing both these factors together has on PIP's performance.

The data in Table 5.8 formed the basis of the significance tests performed to test Hypothesis 1 regarding the proportion of rules that are useful for subsequent problem solving. The first set of two tests compared the means going across the top row of Table 5.8 to evaluate the impact of increasing effect similarity on the proportion of the useful rules learned by PIP. It compared the mean proportion of useful rules in the 30-45 set with the mean proportion of useful rules in the 30-70 set, and the mean proportion of the useful rules in the 30-70 case with the mean proportion of the useful rules by PIP in the 30-80 set. It was found that the mean proportion of rules that prove useful for subsequent planning in the 30-70 set (0.45) is significantly greater than the mean proportion of rules that are useful in the 30-45 case (0.40) [$t = 1.91,\ p < 0.05$].

Similarly, the mean proportion of useful rules in the 30-80 case (0.60) is significantly greater than the mean proportion of useful rules for the 30-70 case (0.45) [$t = 3.90$, $p < 0.05$]. Thus increasing the effect similarity increases the proportion of useful rules learned by PIP.

The second set of tests was aimed at comparing the means going down the leftmost column of Table 5.8 to evaluate the impact of increasing precondition similarity on the proportion of the useful rules learned by PIP. It was found that the mean proportion of rules that are useful for subsequent planning in the 40-45 case (0.50) is significantly greater than the mean proportion of the useful rules learned in the 30-45 case (0.40) [$t = 4.33$, $p < 0.05$]. Similarly, the mean proportion of the rules that are useful in the 50-45 case (0.63) is significantly greater than the mean proportion of the rules that are useful in the 40-45 case (0.50) [$t = 3.56, p < 0.05$]. Thus increasing the precondition similarity increases the proportion of useful rules learned by PIP.

A third set of t-tests compared the means going down along the diagonal of Table 5.8 to evaluate the impact of increasing both precondition and effect similarity on the proportion of useful rules learned by PIP. A t-test found that the mean proportion of rules that are useful for subsequent planning in the more similar 40-70 case (0.50) is significantly greater than the mean proportion of the useful rules learned by PIP in the less similar 30-45 case (0.40) [$t = 4.33$, $p < 0.05$]. Another t-test found that the mean proportion of useful rules in the more similar 50-80 case (0.65) is significantly greater than the mean proportion of useful rules in the 40-70 case (0.50) [$t = 3.41$, $p < 0.05$]. Thus increasing both precondition and effect similarity increases the proportion of useful rules learned by PIP. This is what was predicted by Hypothesis 1.

The data in Table 5.9 formed the basis of the significance tests performed to test Hypothesis 2 regarding the mean proportion of rules that lead to lower quality plans and hence need to be refined. The first set of t-tests compared the means going across the top row of Table 5.9 to evaluate the effect of increasing effect similarity on the proportion of rules needing refinement. It compared the mean proportion of the rules needing refinement for the 30-45 case with the mean proportion of the rules needing refinement for the 30-70 case, and the mean proportion of the rules needing refinement for the 30-70 case with the mean proportion of the rules needing refinement for the 30-80 case. It was found that the mean proportion of the rules needing refinement for the more similar 30-70 case (0.10) is greater than the mean proportion of the rules needing refinement for the less similar the 30-45 case (0.05) [$t = 4.33$, $p < 0.05$]. Similarly, the mean proportion of rules needing refinement in the 30-80 case (0.15) was found to be significantly greater than the mean proportion of the rules needing refinement in the 30-70 case (0.10) [$t = 2.74$, $p < 0.05$]. Thus increasing effect similarity increases the proportion of rules needing refinement.

A second set of tests compared the means going down the leftmost column of Table 5.9 to assess the impact of increasing precondition similarity on the proportion of rules needing refinement. It was found that the mean proportion of the rules needing refinement in the 40-45 case (0.10) is significantly larger than the mean proportion of rules needing refinement in the 30-45 case (0.05) [$t = 2.74$, $p < 0.05$]. Another t-test found that the mean proportion of the rules needing refinement in the 50-45 case (0.15) is significantly larger than the mean proportion of rules needing refinement in the 40-45 case (0.10) [$t = 1.91$, $p < 0.05$]. Thus increasing precondition similarity increases the proportion of rules needing refinement.

A third set of tests compared the means going down along the diagonal of Table 5.9 to evaluate the impact of increasing both the precondition and the effect similarity on the proportion of rules that PIP learns that need to be refined. A t-test found that the mean proportion of rules needing refinement in the more similar 40-70 case (0.10) is significantly larger than the mean proportion of the rules needing refinement in the less similar 30-45 case (0.05) [$t = 2.74$, $p < 0.05$]. Similarly, the mean proportion of the rules needing refinement in the more similar 50-80 case (0.20) is significantly greater than the mean proportion of rules needing refinement in the less similar 40-70 case (0.10) [$t = 2.49$, $p < 0.05$]. Thus increasing both precondition and effect similarity increases the proportion of rules needing refinement. This is what was predicted by Hypothesis 2.

The data in Table 5.10 formed the basis of the significance tests performed to test Hypothesis 3 regarding improvements in plan quality obtained by PIP. The first set of tests compared the means going across the top row of Table 5.10 to evaluate the impact of increasing effect similarity on the mean value of the plan quality metric. It was found that the mean value of the plan quality metric (i.e., the normalized distance from the optimal quality plans) in the 30-70 case (0.70) is significantly better[3] than the mean value of the plan quality metric in the 30-45 case (0.99) [$t = 3.81$, $p < 0.05$]. Another t-test found that the mean value of the plan quality metric (i.e., the normalized distance from the optimal quality plans) in the 30-80 case (0.38) is significantly better than the mean value of the plan quality metric obtained in the 30-70 case (0.70) [$t = 5.04$, $p < 0.05$].

A second set of t-tests compared the means going down the leftmost column of Table 5.10 to assess the impact of increasing precondition similarity on PIP's performance on plan quality. A t-test showed that the mean value of the plan quality metric (i.e., the normalized distance from the optimal quality plans) in the 40-45 case (0.80) is significantly greater than the mean value of the plan quality metric in the 30-45 set (0.99) [$t = 2.42$, $p < 0.05$]. Similarly, the mean value of the plan quality metric (i.e., the normalized distance from the optimal quality plans) in the 50-45 case (0.60) is significantly better than the mean value of the plan quality metric obtained in the 40-45 case (0.80) [$t = 3.27$, $p < 0.05$].

A third set of tests compared the means going down along the diagonal of Table 5.10 to study the impact of increasing both precondition and effect similarity on PIP's performance on plan quality. The mean plan quality value (i.e., the normalized distance from the optimal quality plans) for 40-70 case (0.58) is better than the mean quality value obtained for the 30-45 case (0.99). A t-test found that the difference between the two means ($0.99 - 0.58 = 0.41$) is statistically significant [$t = 5.57$, $p < 0.05$]. However, smaller improvement ($0.58 - 0.42 = 0.16$) in plan quality is obtained as the domain similarity is further increased to 50-80. This means that as both precondition and effect similarity are increased, PIP's performance increases as the domain similarity is increased. However, smaller gains in plan quality are obtained as the domain similarity is further increased. This is what was predicted by Hypothesis 3.

---

[3] Since plan quality metric measures the normalized distance from the optimal quality plans, smaller values of the plan quality metric are better than the larger values.

5.2.5 Varying the Quality Branching Factor

Quality branching factor was varied by changing the distribution of the quality numbers associated with each action. For instance, when the cost values of all domain actions are set to the same value, the quality branching factor becomes 1 regardless of the branching factor. However, when the cost values are all different the quality branching factor may become as large as the branching factor.

Hypothesis 4 *PIP's plan quality improvements will increase as the quality branching factor is increased.*

Experimental set up. A domain was generated using the domain generation algorithm described in Section 5.2.3 by setting the number of initial conditions and goals to 9. The cost values associated the domain actions in this domain were then varied to generate four domain sets such that each domain had a different value of the quality branching factor. The metric used for measuring the quality branching factor was the average number of different quality plans per problem. This number was computed by exhaustively searching for all plans (up to a resource limit) for 10 randomly chosen problems from each domain and computing the average number of different quality plans per problem.

Results. Table 5.11 presents the mean values of the planning efficiency metric (i.e., the number of new nodes generated by PIP) as a function of the quality branching factor. Table 5.12 presents the values of the plan quality metric (i.e., the distance between PIP's plans and the optimal quality plans) obtained by training PIP on 20, 30, 40, and 60 training examples from the four domains with quality branching factors of 6, 12 , 35 and 60. Table 5.13 shows the mean values of the plan quality metric obtained for the 20-problem sets along with the standard deviations between the plan quality values obtained for the six problem sets (corresponding to the six cross-validation runs in the 20-problem case).

Table 5.11 shows that there is little impact of varying quality branching factor on PIP's planning efficiency. However, Tables 5.12-5.13 shows that PIP's performance on plan quality varies greatly as the quality branching factor is changed. Three one-tailed t-tests were performed to assess statistical significance of these differences in PIP's performance. The first test compared the mean value of the plan quality metric for the 6-case (i.e., the domain that has the quality branching factor of six) with the mean plan quality value for the 12-case. The second test compared the mean quality value for the 12-case with the mean quality value for the 35-case, and fourth test compared the mean quality value for the 35-case with the mean quality value for the 60-case. It was found that the mean value of the plan quality metric (i.e., the distance from the model plans) in the 6-case (is significantly larger than the mean value of plan quality metric in the 12-case [$t = 5.18$, $p < 0.05$]. This means that significantly greater improvements in plan quality are obtained in the domain that has a larger quality branching factor. This is what was expected given Hypothesis 4.

The mean value of the quality metric for the 35-case is also lower than the mean value of the quality metric for the 12-case i.e., the quality improvement in the 35-case is larger than the quality improvement obtained in the 12-case. However, the differences between the two means are not statistically significant [$t = 1.12$, $p < 0.05$]. The top row of Table 5.13 shows that even smaller gains in plan quality are obtained as the quality branching factor is increased from 35 to 60. These results appear to suggest that initially increasing the quality branching factor significantly increases PIP's performance (as predicted by Hypothesis 4) but smaller increases in performance are obtained when the quality branching factor is further increased (in apparent contradiction of Hypothesis 4).

In order to understand why that happens, an analysis of how quality

		Number of training examples				
		0	20	30	40	60
Quality Branching Factor	60	29.6	25	23.6	24.8	24.1
	35	27.5	27.2	25.1	25.8	22.6
	12	26	26.1	23.4	23.2	22.4
	6	22.9	21.3	19.4	21.5	18.5

Table 5.11: Mean planning efficiency metric as a function of quality branching factor and training set size.

branching factor affects PIP's performance is required. The main reason why larger improvements in values of the plan quality metric (i.e., $Q_3 = \frac{\textit{distance of the rule learning planner from model plans}}{\textit{distance of the base planner from model plans}}$) are obtained when the quality branching factor is increased, is that the value of the denominator in Equation 4.1 decreases. This happens because as the quality branching factor is decreased the performance of PIP's base planner decreases because the chances of randomly selecting a path that leads to lower quality plan increase (since there are fewer paths). To see that, assume that all planning paths are of length 1 and that there is only one planning path that leads to the optimal quality plans. The chances of randomly selecting a wrong path (i.e., a path leading to a non-optimal plan) are 83% when the quality branching factor is 6, 91% when the quality branching factor is 12, 97% when the quality branching factor is 35, and 98% when the quality branching factor is 60. So the corresponding increases in chances of randomly selecting a wrong plan are 8% as the quality branching factor is increased from 6 to 12, but there is only 1% increase in the chances of randomly selecting a wrong planning path as the quality branching factor is increased from 35 to 60. This may explain why larger improvements in the mean plan quality values are observed when the quality branching factor is increased from 1 to 12 but smaller improvements in mean plan quality values are observed as the quality branching factor is increased from 12 to 35 and then from 35 to 60.

		Number of training examples				
		0	20	30	40	60
Quality Branching Factor	60	1	0.19	0.15	0.18	0.07
	35	1	0.17	0.10	0.14	0.06
	12	1	0.25	0.31	0.24	0.15
	6	1	0.56	0.47	0.46	0.45

Table 5.12: Mean plan quality metric as a function of quality branching factor and training set size.

		Mean	Standard deviation
Quality Branching Factor	60	0.19	0.138
	35	0.17	0.129
	12	0.25	0.118
	6	0.56	0.087

Table 5.13: Mean and standard deviation of the plan quality metric in the 20-problem case as a function of quality branching factor.

5.2.6 Varying the Correlation Between the Planner Biases and the Quality Improving Biases

When PIP's base planner uses DFID as a search strategy, it becomes biased towards producing shorter solutions. The correlation between the planner's bias and the quality improving bias was defined as the relationship between the cost of an action and the number of the effects it adds.

Given the discussion in Section 5.2, I propose the following hypothesis.

Hypothesis 5 *PIP's performance on plan quality metric will show greater improvements as the difference between the planner's bias and the quality bias increases.*

Experimental set-up. To test for the hypothesis regarding the association between the planner's default bias and the quality improving bias, three domains were generated. The longer plans had a higher quality in Domain I while the shorter plans had higher quality in Domain III. In Domain II, the quality numbers were randomly distributed. So the planner bias was positively correlated with the quality bias in Domain I while the planner bias was

		Number of training examples				
		0	20	30	40	60
positive correlation	Dom I	1	0.65	0.54	0.40	0.37
no correlation	Dom II	1	0.33	0.35	0.30	0.18
negative correlation	Dom III	1	0.15	0.08	0.10	0.05

Table 5.14: Mean plan quality metric as a function of bias correlation and training set size.

		Mean	Standard deviation
positive Correlation	Dom I	0.65	0.045
no correlation	Dom II	0.33	0.064
negative correlation	Dom III	0.15	0.077

Table 5.15: Mean and standard deviation of the plan quality metric in the 20-problem case as a function of bias correlation.

negatively correlated with the quality bias in Domain III.

Results. Table 5.14 shows the values of the average plan quality metric (i.e., the distance between PIP's plans and the model plans) obtained by training PIP on 20, 30, 40, and 60 training examples from the three domains obtained by varying the correlation between the planner bias and the quality bias. Table 5.15 shows the mean values of the plan quality metric and the standard deviations between the six value of the plan quality metric (corresponding to the six cross-validation runs) in the 20-problem case.

The data in Table 5.15 formed the basis of the three one-tailed t-tests were performed to assess the impact of varying bias correlation on PIP's performance with respect to plan quality. The first t-test indicated that PIP's performance is significantly better for Domain III (when the two biases are negatively correlated) than Domain I (when the two biases are closely correlated) $t = 13.73$, $p < 0.05$. PIP's performance for Domain II (no systematic correlation) is also significantly better than PIP's performance on Domain III (negative correlation) $t = 4.40$, $p < 0.05$. A third t-test showed that PIP's performance on Domain II (no systematic correlation) is significantly better than PIP's performance on Domain I (positive correlation) $t = 10.02$, $p < 0.05$. This is what was expected given Hypothesis 5.

		Number of training examples				
		0	20	30	40	60
positive correlation	Dom I	21.5	12	10.4	9.6	8.4
no correlation	Dom II	21.5	10.1	9.7	8.2	7.5
negative correlation	Dom III	21.5	9.3	9.4	9.6	9.5

Table 5.16: Mean planning efficiency metric as a function of bias correlation and training set size.

Table 5.16 shows that when longer plans have better quality (i.e., in Domain III), there is little change in the number of new nodes that need to be expanded. This is because the higher quality (and longer) plans require more nodes to be expanded and slow down the savings achieved by replaying previously cached nodes.

Discussion

The systematic variation in PIP's performance observed by varying different domain features makes sense when PIP is viewed as a supervised concept learner. A supervised concept learner is expected to perform well when the testing items are similar to the training items because more of the knowledge learned during the training phase is expected to be applicable in the testing phase. However, when items belonging to different classes are similar to one another, then it is harder for a concept learner to separate them into different classes. This is essentially what the results from the first set of experiments show. As the amount of similarity between the training and testing items is increased, a larger proportion of PIP's rules are useful for subsequent planning. This leads to an improvement in PIP's performance. However, as the similarity between items belonging to different classes also increases, PIP misapplies a larger proportion of its rules. As the proportion of rules being wrongly applied increases, PIP's performance improvements become smaller. This is what was expected because of PIP's formulation as a concept learner.

The second set of experiments shows that PIP learns well when the quality branching factor is large. This is because when the quality branching factor is small there are few successful planning paths to choose from at each decision point. This makes the likelihood of the planner randomly selecting a good planning path high. This means that there is little for PIP to learn when the quality branching factor is small. Results show that PIP performs better in domains with larger quality branching factors. This bodes well for the scalability of PIP's learning techniques because more complicated real world domains also have higher branching factors.

It makes little sense to use a quality learning system such as PIP in domains where plan quality does not matter i.e., where all solutions to a problem have similar quality values. PIP's learning techniques were designed to be used in the domains where problems have multiple solutions of different quality and where the performance of the base planner on plan quality is not satisfactory (i.e., the base planner produces low quality solutions). The second set of experiments shows that PIP performs well in domains where multiple solutions of different quality exist. The third set of experiments shows that PIP also does well in the domains in which the planner's default biases are negatively correlated with PIP's target function. This is good news because it makes little sense to use a quality learning system such as PIP in the domains where the base planner can produce good quality solutions for most problems without any learning. It is in the domains where the base planner produces low quality plans that a system that can learn to improve plan quality is required. Hence the empirical results demonstrate that PIP performs well in the type of domains for which it was designed.

5.3 Summary

This chapter describes experiments done to empirically evaluate PIP's performance. The results are analyzed and explained by demonstrating that PIP's learning component (ISL) is a supervised concept learner. ISL uses analytic techniques to learn which planning decisions to apply to which partial plans in order to produce high quality solutions. The results of the experiments reported in the first section of this chapter show that PIP needs fewer examples to learn to improve plan quality than an inductive planning and learning system called SCOPE. The second part of this chapter shows that the improvements in plan quality obtained by PIP are affected by a number of domain features including the domain and problem similarity, the quality branching factor and the difference between PIP's default biases and the quality improving biases. The results confirm that PIP performs well in domains for which it was designed i.e., the domains in which (a) multiple solutions exist and some are more preferable than the others, and (b) the planner does not produce the preferable solutions without any learning.

Chapter 6

Conclusions and Future Work

> It is a bad plan that admits of no modification.
> (Publius Syrus as quoted by [Wil96]).

Being able to efficiently produce good quality solutions is essential if AI planners are to be widely applied to the real-world situations. However, conventional wisdom in AI has been that "domain independent planning is a hard combinatorial problem. Taking into account plan quality makes the task even more difficult" [AK97]. This dissertation has presented a novel technique for learning domain specific heuristics for partial order planners that improves plan quality without sacrificing much in the way of planning efficiency. PIP's learning algorithm is also analyzed as a supervised concept learner that learns to discriminate between the partial plans it encounters during the search and learns to apply the appropriate planning decisions (i.e., the planning decisions that will lead towards a higher quality plan) to a partial plan.

PIP's learning module, ISL, compares two different quality planning episodes to compute search control rules that improve plan quality. The principal limitation of the PIP approach is the assumption that the plan quality knowledge can be encoded into a value function[1]. The planning episode is a trace of the planning decisions taken to produce a plan. The basic idea is to compare the planning decisions that lead to a low quality plan with the planning decisions that lead to a higher quality plan to compute rules that guide the planner towards following better planning decisions for similar subsequent problems.

---

[1] The current version of PIP is more limited than that. Even in certain situations where plan quality can be encoded into a value function (such as a transportation domain in which quality of a plan depends on which truck or plane is used for transportation), it is unclear how PIP can learn its quality improving rules. This issue is discussed further in Section 6.2.2.

This framework supports two approaches to improving plan quality. The first is the traditional approach of learning search control rules and using them during the search to bias the planner towards higher quality solutions. The second is the more recent planning by rewriting approach, which involves first generating a complete plan and then rewriting it into a higher quality plan. Experiments done in several benchmark planning domains show that both these approaches lead to higher quality plans, but using search control rules is more efficient than using plan rewrite rules. This is the approach that is then adopted as the main PIP approach and compared to SCOPE, the only other planning and learning system that learns quality improving search control rules for partial-order planners. The experimental results show that PIP's search control rules can improve plan quality (measured as plan length) using fewer training examples than SCOPE.

The PIP approach to learning search control rules can also be seen as a technique for learning the rationales for applying various planning decisions. Plan rationale has been variously defined as "why the plan is the way it is", and as "the reason as to why the planning decisions were taken" [PT98]. The usefulness of storing plan rationale to help future planning has been demonstrated by various case-based planning approaches such as PRODIGY/ANALOGY [Vel94], DerSNLP [IK97] and [VMM97]. However, the previous techniques are unable to distinguish between planning decisions that, while leading to successful plans, may produce plans that differ in quality. PIP uses a richer language for representing planning rationales that allows it to learn to distinguish between such planning decisions.

6.1 Major Contributions of This Work

1. This work is the first comprehensive study of the problem of learning to improve plan quality for partial order planners. The result of this study is a framework for (a) comparing planning episodes that lead to two plans

of different quality for a problem and (b) extracting information about their differences and storing this information in a rule form. These rules can be of two different kinds: search control rules and plan rewriting rules.

2. A system that stores the information derived by comparing the two planning episodes as search control rules was designed and implemented. PIP's performance was measured by testing it on both benchmark planning domains and systematically designed artificial domains. The results show that PIP performs well on the domains for which it was designed such as the domains where problems have multiple solutions of different quality and the base planner is unable to produce high quality solutions for most problems. The empirical investigation also help us understand PIP's limitations: it is limited by the factors that limit the performance of all supervised concept learning systems such as the requirement that the training and testing examples be drawn from identical populations.

   How should one decide whether to use PIP, versus some other quality learning system? PIP is clearly the only choice when the objective is to learn to improve multi-attribute quality function and no domain experts are available to provide better quality plans. Improvements in plan quality obtained by PIP are the largest in domains where each problem has a number of solutions of different quality and the default planner does not produce high quality solutions for most problems.

3. Another planning and learning system that learns plan-rewrite rules and then uses them to improve the quality of its plans was also designed and implemented. PIP-rewrite was tested on several benchmark domains to show that plan-rewrite rules *can be* learned automatically.

   Recall that all previous planning by rewriting systems used manually encoded rewrite rules. My main interest in rewrite rules was not to find the best way of learning rewrite rules automatically but (a) to see if PIP's techniques can also be used to learn rewrite rules, and (b) to evaluate the benefits and costs of using the results of PIP's techniques to learn plan-rewrite rules instead of learning search control rules. In order to do this both search control and rewrite rule were learned using PIP's learning component and then the performance of the planner that used rewrite rules was compared with the performance of the planner that used search control rules. The results show that there is information to be gained by following PIP's technique of analyzing local refinement decisions during the planning process and translating them into rewrite rules. However, it is harder to learn good rewrite rules by following PIP's standard learning algorithm than learning good search control rules. This is partly because of the inherent difficulty of translating the information learned from one context (i.e., the context of choosing between plan refinement paths) into a form usable in another context (i.e., replacing portions of complete plans). Another problem is that ISL was really designed to learn search control rules and many of the design decisions do not make sense for learning rewrite rules. For instance, PIP's practice of discovering subsequent conflicting choice points (after discovering the first conflicting choice point) is harder to justify when the objective is to learn rewrite rules. The shortcomings of PIP's approach to learning rewrite rule are presented (here and in more detail in Chapter 4) in the hope that by showing a sub-optimal way of learning rewrite rules, this work will motivate others to improve on this techniques.

6.2 Future Research Directions

There are a number of ways in which PIP can be improved and extended. Improvements to PIP include better organization of the rule library and extensions of PIP include extending PIP's learning techniques for the non-classical planners.

6.2.1 Better Rule Organization

Currently PIP's rule library is not organized. This means that the search process for retrieving the rule takes considerable time. This especially becomes a problem as the number of rules increases. One idea is to organize the rule library as a hierarchy with the most general rules at the top and the most specific rules at the bottom of the hierarchy. Rule data presented for PIP in Chapter 4 and 5 also suggests that not all of PIP's rules are useful for subsequent problem solving. This suggests doing a rule utility analysis [Min89] to keep track of the cost of keeping a rule around versus the potential loss of information accrued by forgetting it. This analysis could then be used to forget the rules that are not very useful. This can potentially reduce the size of the rule library and improve the rule retrieval time.

Another reason why PIP's rule library expands so rapidly is that two rules are considered different even if all the planning decisions in their trace field are the same and only difference between them is the goals that they solve. This happens when the same planning decisions can be used to solve two different training problems because some actions added by the planning decisions have multiple available effects. As mentioned in Section 4.2.5, a modification to PIP's rule storing algorithm to store a planning decision sequence by all the goals it can possibly resolve (even though it may not have been used to solve all of them in the example problem from which the rule was learned) can help. This modification would not only get more mileage out of a planning decision sequence (and improve PIP's performance on domains such as Softbot) but also reduce the size of PIP's rule library However, it may increase the number of rules being wrongly retrieved.

I would like to perform a careful evaluation of the benefits and costs of both rule organization strategies suggested here.

```
Action: move-briefcase(Bcase From To)
        precondition: {at(Bcase, From), neq(From, To)}
        effect: {at(Bcase,To), not(at(Bcase, From)),
                 in(P, Bcase) -> {at(P, To), not(at(P, From))}
                 in(D) -> {at(D, To), not(at(D, From))}}
```

Figure 6.1: Move-briefcase action from Pednault's Briefcase Domain.

6.2.2 Extending PIP To Deal With More Expressive Languages

Any STRIPS domain in which plan quality knowledge can be expressed as a value function can be encoded into PR-STRIPS. However, the current version of PIP cannot learn from all such domains. For instance, domains in which quality of a plan depends on the use of a particular resource such as a transportation domain in which quality of a plan depends on which truck or plane is used for transportation. The current version of PIP cannot learn effective quality improving rules in such domains. I would like to investigate what changes are required to PIP's learning techniques to learn quality improving rules in such situations.

A direction along which PIP's planner could be extended is to deal with with more expressive languages than STRIPS such as Pednault's ADL [Ped89] which allows reasoning with quantified preconditions and conditional effects. The standard approach to extending the partial order planning framework (such as the approach taken by UCPOP) makes few changes to the planning decisions themselves. For instance, conditional effects such as those used by Pednault's briefcase domain (shown in Figure 6.1) can be easily handled by a STRIPS planner by treating conditions of the effects that are required by some action as action preconditions. I expect that extending PIP for ADL will require minimum changes in PIP's learning techniques because PIP's learning framework only depends on the planning decisions and not on the planning language used as such.

6.2.3 Combining ISL With EBL From Failures

PIP is similar to DerSNLP insofar as both learn only from planning successes and do not learn from planning failures. If PIP is viewed as a planning and learning system that learns rationales for planning decisions, then not learning anything from planning failures seems like a loss of learning opportunities. PIP does not learn the rationale for the planning decisions sequences that can lead to failures. Ihrig and Kambhampati [IK97] show that DerSNLP+EBL by learning from both planning successes and failures can improve its performance more than it can using either of the two techniques alone.

It may be possible to prevent PIP from going down a number of dead ends if it were to learn SNLP+EBL like search control rules from failures. I believe that this can reduce backtracking in PIP considerably and lead to significant improvements in PIP's planning efficiency.

6.2.4 Extending PIP's Techniques For Non-classical AI Planners

Kambhampati *et al.* [KPL97] describe how various approaches to planning can be described as variations on the refinement planning framework. Their generalized planning algorithm has two main phases: refinement and solution extraction. While classical planners (such as partial order planners) spend most of their planning effort in the refinement phase, the solution extraction phase dominates the complexity of the newer approaches to planning (such as Graphplan). This may partly explain why most of the learning techniques for Graphplan focus on the solution extraction phase. Kambhampati [Kam00] shows how UCPOP+EBL's [KKQ96] EBL techniques can be extended for Graphplan to determine the *necessary* constraints responsible for the failure of a node.

Unfortunately, the EBL approach cannot be used to distinguish between two planning decisions both of which lead to valid plans of differing quality, because there is no planning failure to reason about. Consider the planning graph shown in figure 6.2 in which the solid straight lines show the action-

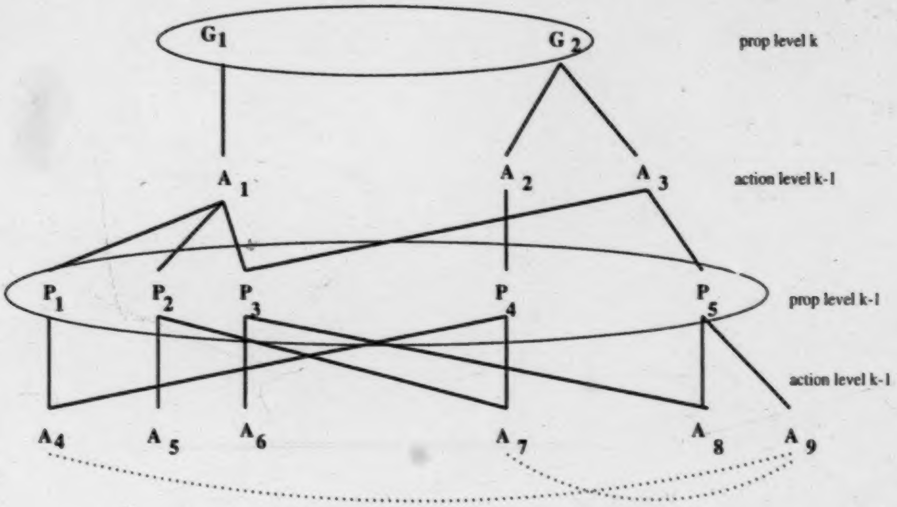


Figure 6.2: A planning graph.

precondition/effect relationship and the broken curved lines show the *mutex* relations. Two actions are said to be mutex if they cannot possibly happen at the same time (e.g., if one of them deletes a precondition/effect of the other action). Suppose that the plan $P_1 = \{A_1, A_4, A_5, A_8, A_3\}$ has a higher quality than the plan $P2 = \{A_1, A_4, A_7, A_6, A_2\}$ according to some user-specified quality function q i.e., $q(P_1) > q(P_2)$. By analyzing the backward phase of the two planners that produced these different plans, it can be seen that the choices at the conflicting choice points (namely, evaluation of the propositions G_2 and P_2) are *responsible* for the varying qualities of the two plans. The search-control rule that can be learned from this episode is:

```
if q([A2,A7]) > q([A3,A9]) in the context of the current problem
    then
         assign the value A2 to G2 and A7 to P4
    else
         assign the value A3 to G2 and A8 to P5.
```

I would like to investigate if learning rules such as this can help Graphplan improve the quality the plans it produces.

6.2.5 Extending PIP-rewrite's Techniques For Non Classical AI Planners

As pointed out in Section 2.3.4, for planning by rewriting systems, it does not matter how the initial plan was generated. One obvious idea for improving PIP-rewrite's planning efficiency on test problems is to train PIP-rewrite by running a POP-based planner but then create the initial plan for all the test problems using a state of the art planner such as Graphplan [BF97] or Blackbox [KS98].

Yet another idea is to use a state of the art planner such as Graphplan[BF97] or Blackbox throughout the training and testing phase for the rewrite-rule learner. The easiest way to extend the PIP-rewrite's techniques would be to use the PIP's constraint-inference mechanism to infer the ordering constraints for both the system's plan (and not just for the model plan) and using that to learn the rewrite rules. I believe that this implies only trivial modifications to PIP-rewrite's learning mechanism and can considerably improve PIP-rewrite's performance on planning efficiency.

6.3 Summary

This dissertation has presented a novel technique for automatically learning and incorporating domain specific knowledge into domain independent partial order planners. What sets this technique apart from other techniques for learning to improve plan quality is that the plan quality knowledge forms an essential part of the rules. This novel approach to learning quality improving heuristics opens up a number of research areas. This chapter has outlined some of those ideas. I believe that work along these lines will contribute towards making AI planning applicable to more practical planning situations.

Bibliography

[AHT90] J. Allen, J. Hendler, and A. Tate. *Readings in Planning.* Morgan Kaufmann, San Mateo, CA, 1990.

[AK97] J. Ambite and C. Knoblock. Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 706–713, Menlo Park, CA, 1997. AAAI Press.

[AK98] J. Ambite and C. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 3–10, Menlo Park, CA, 1998. AAAI Press.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 15:281–300, 1997.

[BN98] F. Baadr and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

[BV94] D. Borrajo and M. Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning*, pages 64–82, Berlin, 1994. Springer Verlag.

[BW94] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[DGT95] B. Drabble, Y. Gil, and A. Tate. Acquiring criteria for plan quality control. In *Notes of the AAAI Spring Symposium on Integrated Planning Applications*, pages 36–40, 1995. AAAI Tech Report SS-95-04.

[dH90] G. de Soete and F. Hubert. *New Developments in Psychological Choice Modeling.* North-Holland, New York, 1990.

[Dic73] C. Dickens. *Great Expectations.* J.R. Osgood, Boston, MA, 1873.

[EM97] T. Estlin and R. Mooney. Learning to improve both efficiency and quality of planning. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1227–1233, Los Altos, CA, 1997. Morgan Kaufmann.

[ENS95] K. Erol, D. Nau, and V. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76:75–88, 1995.

[Est98] T. Estlin. Using multistrategy learning to improve planning efficiency and quality. Technical Report AI98-269, PhD Thesis, University of Texas at Austin, 1998.

[EW94] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *Communications of the ACM*, 37:72–76, 1994.

[Fis70] P. Fishburn. *Utility Theory for Decision Making.* Wiley, New York, 1970.

[FN71] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[FS75] J. Feldman and R. Sproul. Decision theory and artificial intelligence II: The hungry monkey. *Cognitive Science*, 1:158–192, 1975.

[GS96] A. Gervini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[Ham90] K. Hammond. Case-based planning: A framework for planning from experience. *Cognitive Science*, 14:385–443, 1990.

[Hen81] D. Hensher. *Applied Discrete-choice Modeling.* Wiley, New York, 1981.

[HH94] P. Haddawy and S. Hanks. Decision-theoretic refinement planning using inheritance abstraction. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 266–271, 1994.

[HH98] P. Haddawy and S. Hanks. Utility models for goal-directed, decision-theoretic planners. *Computational Intelligence*, 14:392–429, 1998.

[Ihr96] L. Ihrig. The design and implementation of a case-based planning framework within a partial order planner. Technical Report ASU-CSE-96-007, PhD thesis, Department of Computer Science, Arizona State University, 1996.

[IK97] L. Ihrig and S. Kambhampati. Storing and indexing plan derivations through explanation-based analysis of retrieval failures. *Journal of Artificial Intelligence Research*, 7:161–198, 1997.

[Iwa94] M. Iwamoto. A planner with quality goal and its speed up learning for optimization problems. In *Proceedings of Second International Conference on AI Planning Systems*, pages 281–286, 1994.

[Kam00] S. Kambhampati. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of Artificial Intelligence Research*, 12:1–34, 2000.

[Kel87] R. M. Keller. Concept learning in context. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 91–102, Los Altos, 1987. Morgan Kaufmann.

[KKQ96] S. Kambhampati, S. Katukam, and Y. Qu. Failure driven dynamic search control for partial order planners. *Artificial Intelligence*, 88:253–316, 1996.

[Kno96] C. Knoblock. Building a planner for information gathering: A report from the trenches. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 134–141, Menlo Park, CA, 1996. AAAI Press.

[Koe98] J. Koehler. Planning under resource constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 489–493, Berlin, 1998. Springer Verlag.

[Kol93] J. Kolodner. *Case Based Reasoning.* Morgan Kaufmann, San Mateo, CA, 1993.

[Kor85] R. Korf. Depth-first iterative-deeping: An optimal admissible tree search. *Artificial Intelligence*, 27:97–110, 1985.

[KPL97] S. Kambhampati, E. Parker, and E. Lambrecht. Understanding and extending graphplan. In *Proceedings of the Fourth European Conference on Planning*, pages 260–266, 1997.

[KR93] R. Keeney and H. Raiffa. *Decisions With Multiple Objectives: Preferences and Value Tradeoffs.* Cambridge University Press, New York, 2nd edition, 1993.

[KS98] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of Fourth International Conference on Artificial Intelligence Planning Systems*, pages 181–189, Menlo Park, CA, 1998. AAAI Press.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 318–325, Los Altos, CA, 1999. Morgan Kaufmann.

[KV94] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, 1994.

[Lou88] J Louviere. *Analyzing Decision Making: Metric Conjoint Analysis.* Sage Publications, Newbury Park, 1988.

[Min89] S. Minton. Explanation-based learning. *Artificial Intelligence*, 40:63–118, 1989.

[Mit83] T. Mitchell. Learning and problem solving. In *Proceedings of Eighth International Joint Conference on Artificial Intelligence*, Los Altos, CA, 1139–1151 1983. Morgan Kaufmann.

[MKK86] T. Mitchell, R. Keller, and S. Keddar-Cabelli. Explanation based learning: A unifying view. *Machine Learning*, 1:47–80, 1986.

[MMST93] S. Mahadevan, T. Mitchell, L. Steinberg, and P. Tadepalli. An apprentice-based approach to knowledge acquisition. *Artificial Intelligence*, 64:1–52, 1993.

[MR91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Ninth National Conference on Artificial Intelligence*, pages 634–639, Menlo Park, CA, 1991. AAAI Press/MIT Press.

[MR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.

[NS72] A. Newell and H. Simon. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Ped89] E. Pednault. Exploring the middle ground between STRIPS and the situation calculus. In *Principles of Knowledge Representation and Reasoning*, pages 324–332. Morgan Kaufmann, Los Altos, CA, 1989.

[Per96] A. Perez. Representing and learning quality-improving search control knowledge. In *Proceedings of the 13th International Conference on Machine Learning*, pages 382–390, Los Altos, CA, 1996. Morgan Kaufmann.

[PT98] S. Polyak and A. Tate. Rationale in planning: Causality, dependencies, and decisions. *Knowledge Engineering Review*, 13:1–16, 1998.

[RK93] D. Ruby and D. Kibler. Learning steppingstones for problem solving. *International Journal of Pattern Recognition and Artificial Intelligence*, 7:527–540, 1993.

[Sac74] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sus73] G. Sussman. A computational model of skill acquisition. Technical Report AI-TR-297, MIT AI Lab, 1973.

[Tat74] A. Tate. INTERPLAN: A plan generation system which can deal with interactions between goals. Technical Report MIP-R-109, University of Edinburgh, 1974.

[Tat77] A. Tate. Generating project networks. In *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, Cambridge, MA, 1977. IJCAI.

[UE98] M. A. Upal and R. Elio. Learning to improve quality of the plans produced by partial order planners. In *Workshop Notes of the AIPS-98 Workshop on Knowledge Engineering and Acquisition* *for Planning: Bridging Theory and Practice*, pages 94–103, 1998. AAAI Tech Report WS-98-03.

[VCP+95] M. Veloso, J. Carbonell, M. Perez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:81–120, 1995.

[Vel94] M. Veloso. *Learning by Analogical Reasoning.* Springer Verlag, Berlin, 1994.

[VMM97] M. Veloso, A. Mulvehil, and Cox M. Rationale-supported mixed-initiative case-based planning. In *Proceedings of National Conference on Innovative Applications of Artificial Intelligence*, pages 1072–1077, Menlo Park, CA, 1997. AAAI Press.

[Wal77] R. Waldinger. Achieving several goals simultaneously. *Machine Intelligence*, 8:94–136, 1977.

[Wel93] M Wellman. Challenges of decision-theoretic planning: The classical approach and beyond. In *Notes of the AAAI Spring Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond*, pages 156–160, 1993. AAAI Tech Report SS-93-03.

[Wil88] D. C. Wilkins. Knowledge base refinement using apprenticeship learning techniques. In *Proceedings of the National Conference on Artificial Intelligence*, pages 646–651, Menlo Park, CA, 1988. AAAI Press.

[Wil96] M. Williamson. A value-directed approach to planning. Technical Report TR-96-06-03, PhD thesis, University of Washington, 1996.

[Yu85] P. Yu. *Multiple-criteria Decision Making: Concepts, Techniques, and Extensions.* Plenum Press, New York, 1985.

Appendix A

PR-STRIPS Encoding of Transportation Domain

```
Action: load_truck(Obj,Truck,Loc),
    Preconditions: {at_obj(Obj,Loc),at_truck(Truck,Loc)},
    Effects: {inside_truck(Obj,Truck),not(at_obj(Obj,Loc)),
             time(-5), money(-5)}

Action: unload_truck(Obj,Truck,Loc),
    Preconditions: {inside_truck(Obj,Truck),
                   at_truck(Truck,Loc)},
    Effects: {at_obj(Obj,Loc),not(inside_truck(Obj,Truck)),
             time(-5), money(-5)},

Action: drive_truck(Truck,Loc_from,Loc_to),
    Preconditions: {same_city(Loc_from,Loc_to),
                   at_truck(Truck,Loc_from)},
    Effects: {at_truck(Truck,Loc_to),
             not(at_truck(Truck,Loc_from)),
             time(-distance(Loc_from,Loc_to)/50),
             money(distance(Loc_from, Loc_to)/50)},

Action: fly_airplane(Airplane,Loc_from,Loc_to),
    Preconditions: {airport(Loc_to),neq(Loc_from,Loc_to),
       at_airplane(Airplane,Loc_from)},
    Effects: {at_airplane(Airplane,Loc_to),
                 not(at_airplane(Airplane,Loc_from)),
                 time(-distance(Loc_from,Loc_to)/1000),
                 money(distance(Loc_from, Loc_to)/5)},

Action: drive_truck_acities(Truck,Loc_from,Loc_to),
    Preconditions: {at_truck(Truck,Loc_from),
                       neq(Loc_from, Loc_to)},
    Effects: {at_truck(Truck,Loc_to),
                 not(at_truck(Truck,Loc_from)),
                 time(-distance(Loc_from, Loc_to)/50),
                 money(distance(Loc_from, Loc_to)/50)},

Action: unload_airplane(Obj,Airplane,Loc),
    Preconditions: {inside_airplane(Obj,Airplane),
                       at_airplane(Airplane,Loc)},
    Effects: {at_obj(Obj,Loc),not(inside_airplane(Obj,Airplane)),
                 time(-20), money(-15)},

Action: load_airplane(Obj,Airplane,Loc),
    Preconditions: {at_obj(Obj,Loc),at_airplane(Airplane,Loc)},
    Effects: {inside_airplane(Obj,Airplane),not(at_obj(Obj,Loc)),
                 time(-20), money(-15)},

distance(From, To) = |position(From) - position(To)|
```

$$Quality(P = \{a_1, a_2, \ldots, a_n\}) = \textstyle\sum_{i=1}^{n} 5 \times time(a_i) - money(a_i).$$

Appendix B

PR-STRIPS Encoding of Softbot Domain

```
Action:   finger(Person)
    Preconditions: {know_email(Person),
                       has_plan_file(Person)},
    Effects: {know_name(Person), know_address(Person),
                 know_phone(Person), know_inst(Person),
                 time(-1), money(0), help(0), bother(0)}

Action:   netfind(Person)
      Preconditions: {know_name(Person), know_inst(Person)},
      Effects: {know_email(Person), time(-5), money(-1),
                   help(0), bother(0)}

Action:   bibsearch(Person)
     Preconditions: {know_name(Person), published(Person)},
     Effects: {know_inst(Person), time(-2), money(0),
                  help(0), bother(0)}

Action:   homepage_finder(Person)
       Preconditions: {know_name(Person),
                          has_homepage(Person)},
       Effects: {know_email(Person), know_inst(Person),
                    time(-3), money(-1), help(0), bother(0)}

Action:   ask_other_email(Person, Helper)
       Preconditions: {know_name(Person), know_email(Helper),
                             knows_about(Helper, Person)},
       Effects: {know_email(Person), time(-10), money(0),
                    help(-5), bother(0)}

Action:   ask_other_all(Person, Helper)
      Preconditions: {know_name(Person), know_email(Helper),
                            knows_about(Helper, Person)},
         Effects: {know_email(Person), know_address(Person),
                      know_phone(Person), time(-12), money(0),
                      help(-7), bother(0)}

Action:   ask_person_ssn(Person),
      Preconditions: {know_name(Person), know_email(Person)},
         Effects: {know_ssn(Person), time(-10), money(0),
                      help(-5), bother(-1)}

Action:   hire_cyberdetective(Person),
     Preconditions: {know_name(Person)},
     Effects: {know_email(Person), know_inst(Person),
                  know_address(Person), know_phone(Person),
                  know_ssn(Person), time(-120), money(-20),
                  help(0), bother(0)}
```

$$Quality(P = \{a_1, a_2, \ldots, a_n\}) = \sum_{i=1}^{n} time(a_i) + money(a_i) + help_u sed(a_i) + bother_u sed(a_i).$$

Appendix C

PR-STRIPS Encoding of Manufacturing Process-planning Domain

```
Action: weld(X, Y, New_object, Orientation)
    Preconditions: {is_object(X), is_object(Y),
          machine(welder),
          composite_object(New_object, Orientation, X, Y),
          can_be_welded(X,Y, Orientation)},
    Effects: {temperature(New_object, hot),
          joined(X,Y,Orientation),
          not(is_object(X)),not(is_object(Y)), cost(-70)}.

Action: bolt(X, Y, New_object, Orientation, Width)
    Preconditions: {is_object(X), is_object(Y), machine(bolter),
           composite_object(New_object, Orientation, X, Y),
           has_hole(X, Width, Orientation),
           has_hole(Y, Width, Orientation),
           bolt_width(Width), can_be_bolted(X, Y, Orientation)},
    Effects: {not(is_object(X)),not(is_object(Y)),
           joined(X,Y,Orientation),cost(-20)}.

Action: drill_press(X, Width, Orientation)
```

```
    Preconditions: {machine(drill_press), is_object(X),
           is_drillable(X, Orientation), have_bit(Width)},
    Effects: {has_hole(X, Width, Orientation), cost(-50)}.

Action: spray_paint(X, Color, Shape)
    Preconditions: {machine(spray_painter), is_object(X),
           sprayable(Color), temperature(X,cold),
           regular_shape(Shape), shape(X, Shape),
           has_clamp(spray_painter)},
    Effects: {painted(X, Color), cost(-15)}.

Action: immersion_paint(X, Color)
    Preconditions: {machine(immersion_painter),is_object(X),
           have_paint_for_immersion(Color)},
    Effects: {painted(X, Color), cost(-10)}.

Action: punch(X, Width, Orientation)
    Preconditions: {machine(punch), is_object(X),
           is_punchable(X, Width, Orientation),
           has_clamp(punch)},
    Effects: {surface_condition(X,rough),
           has_hole(X, Width, Orientation), cost(-40)}.

Action: grind(X)
    Preconditions: {machine(grinder), is_object(X)},
    Effects: {surface_condition(X, smooth), cost(-30)}

Action: lathe(X)
    Preconditions: {machine(lathe), is_object(X)},
    Effects: {surface_condition(X,rough),shape(X, cylindrical),
  cost(-25)}.
```

```
Action: roll(X)
    Preconditions: {machine(roller), is_object(X)},
    Effects: {temperature(X,hot), shape(X, cylindrical),
           cost(- 20)}.

Action: polish(X)
    Preconditions: {machine(polisher),is_object(X)},
    Effects: {surface_condition(X, polished), cost(-15)}.
```

$$Quality(Plan = \{a_1, a_2, \ldots, a_n\}) = \frac{1}{\sum_{i=1}^{n} cost(a_i)}.$$

Appendix D

An Abstract Domain

```
Action: a1,
        Preconditions: {p1,p2},
        Effects: {g1,g2,g3,g4,g5,
                     cost(-78)}.

Action: a2,
        Preconditions: {p2,p8},
        Effects: {g1,g2,g3,g4,g12,
                     cost(-159)}.

Action: a3,
        Preconditions: {p1,p2},
        Effects: {g2,g3,g4,g5,g11,
                     cost(-39)}.

Action: a4,
        Preconditions: {p3,p4},
        Effects: {g1,g3,g4,g5,g10,
                     cost(-153)}.

Action: a5,
        Preconditions: {p5,p6},
        Effects: {g3,g6,g7,g8,g9,
                    cost(-110


Action: a6,
        Preconditions: {p1,p2},
        Effects: {g6,g9,g8,g10, g12,
                    cost(-125)}.



Action: a7,
        Preconditions: {q1,q2},
        Effects: {p1,p2,p3,p4,p7,
                    cost(-102)}.


Action: a8,
        Preconditions: {q3,q6},
        Effects: {p1,p3,p4,p5,p6,
                    cost(-183)}.


Action: a9,
        Preconditions: {q2,q4},
        Effects: {p2,p3,p5,p6,p8,
                    cost(-143)}.


Action: a10,
        Preconditions: {q1,q5},
        Effects: {p3,p4,p5,p7,p9,
                    cost(-121)}.


Action: a11,
        Preconditions: {q8,q11},
        Effects: {p3,p5,p6,p9,p12,
                cost(-48)}.

Action: a12,
        Preconditions: {q7,q12},
        Effects: {p6,p7,p10,p11,p12,
                cost(-160)}.


Action: a13,
        Preconditions: {i1,i2},
        Effects: {q1,q2,q3,q4,q7,
                cost(-80)}.

Action: a14,
        Preconditions: {i10,i12},
        Effects: {q2,q3,q4,q5,q6,
                cost(-21)}.

Action: a15,
        Preconditions: {i3,i6},
        Effects: {q2,q3,q7,q8,q9,
                cost(-43)}.

Action: a16,
        Preconditions: {i2,i4},
        Effects: {q4,q5,q10,q11,q12,
                cost(-167)}.

Action: a17,
        Preconditions: {i1,i5},
        Effects: {q2,q4,q5,q7,q9,
                cost(-59)}.
```

```
Action: a18,
       Preconditions: {i7,i8},
       Effects: {q1,q2,q3,q4,q8,
                 cost(-104)}.
```

$$Quality(Plan = \{a_1, a_2, \ldots, a_n\}) = \frac{1}{\sum_{i=1}^{n} cost(a_i)}.$$